PARALLEL ALGORITHMS FOR ENABLING FAST AND SCALABLE ANALYSIS OF HIGH-THROUGHPUT SEQUENCING DATASETS

A Dissertation
Presented to
The Academic Faculty

By

Nagakishore Jammula

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2019

PARALLEL ALGORITHMS FOR ENABLING FAST AND SCALABLE ANALYSIS OF HIGH-THROUGHPUT SEQUENCING DATASETS

Approved by:

Dr. Srinivas Aluru, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Moinuddin Qureshi
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Linda Wills
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Date Approved: November 29, 2018

ACKNOWLEDGEMENTS

I am thankful to many people for helping me accomplish what I have through this dissertation.

My advisor, Srinivas Aluru, for having faith in me, for helping me conduct good research and communicate it well, and for offering me feedback to keep getting better.

My committee members - Moinuddin Qureshi, Richard Vuduc, Linda Wills, and Ada Gavrilovska - for helping me improve the quality of dissertation. Moinuddin Qureshi spent a lot of time guiding me through my first publication. Several faculty members supported me during my time at Georgia Tech - Muhannad Bakir, Benny Bing, Jongman Kim, and Sudhakar Yalamanchili. Sriram Chockalingam contributed to publications that formed the basis for this thesis.

My teachers and mentors taught a number of invaluable lessons over the years. A special mention to Martin Kuhlmann, Kamlesh Pandey, Ramprasad C, Veerendra G, and Ramakrishna B.

My family members for offering me unwavering support through the years - Jhansi Lakshmi, Uma Maheswari, Srinivasa Rao, Padma, Koteswara Rao M, Girija, Uma, Lakshmi Phani, Raja Swapna, Nagesh Babu, Sai Ramya, Nageswara Rao, Sambasiva Rao, Sirisha, Nagendramma, Satyanarayana, Koteswara Rao G, Suhas, Madhurima, Santhi Priya, Sudhakar Babu, Sneha Latha, Koteswara Rao J, and Manoj P.

My friends for their camaraderie over the years. A special shout out to former and current members of my research group at Georgia Tech.

The staff at Georgia Tech for their assistance with administrative functions - Jacqueline Trappier, Tasha Torrence, Daniela Staiculescu, Arelene Washington, Carolyn Young, Nirvana Edwards, Rebecca Wilson, and Charmain Alston. I am grateful to National Science Foundation, BITS-Pilani, and IOCL for offering financial support to pursue my academic goals. Finally, I apologize for missing out on recognizing anyone.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

State-of-the-art high-throughput sequencing (HTS) instruments decipher billions of short genomic fragments per run. The output sequences are referred to as *reads*. The objective of this thesis is to develop parallel algorithms for enabling fast and scalable analysis of large-scale HTS short read datasets. Specifically, the thesis tackles the following problems.

Reads contain errors due to limitations of sequencing technology. Read error correction enhances the quality of results produced by applications in areas such as genomics, metagenomics, and transcriptomics. Use of error corrected reads also improves the runtime and the memory usage of such applications. Sequential error correction tools cannot cope with the large number of short reads produced by modern day sequencing instruments. A distributed-memory Parallel Spectrum-based Error Correction (PSbEC) algorithm was proposed to overcome this drawback [1]. In the first part of this thesis, we propose techniques to address three major shortcomings of the PSbEC algorithm. Our optimizations enhance the scope and the speedup of the PSbEC algorithm, thereby enabling error correction of big genomic datasets. More specifically, by combining our optimizations, we are able to achieve a cumulative speedup of up to 11 X. Further, we demonstrate error correction of a human dataset containing nearly 1.55 billion reads. This work is the first demonstration of distributed-memory genomic read error correction for a dataset consisting of more than a billion reads.

HTS short read datasets facilitate a wide variety of analyses with applications in areas such as genomics, metagenomics, and transcriptomics. Owing to the large size of the read datasets, such analyses are often compute and memory intensive. In the second part of this thesis, we present a parallel algorithm for partitioning large-scale read datasets in order to facilitate distributed-memory parallel analyses. During the process of partitioning the read datasets, we construct and partition the associated de Bruijn graph in parallel. This allows

applications that make use of a variant of the de Bruijn graph, such as de novo assembly, to directly leverage the generated de Bruijn graph partitions. In addition, we propose a mechanism for evaluating the quality of the generated partitions of reads and demonstrate that our algorithm produces high quality partitions.

Transmission, storage, and archival of HTS short read datasets pose significant challenges due to the large size of such datasets. Constant improvements to HTS technology, in the form of increasing throughput and decreasing cost, and its increasing adoption amplify the problem. General-purpose compression algorithms have been widely adopted for representing read datasets in a compact form. However, they are unable to fully leverage the domain-specific properties of read datasets. In response, researchers proposed special-purpose compression algorithms which improve upon the compression efficiency of general-purpose compression algorithms. In the last part of this thesis, we present ParRefCom, a parallel *reference-based* algorithm for compressing HTS genomics read datasets. In contrast to existing special-purpose compression algorithms, ParRefCom treats *paired-end* reads as first-class citizens. HTS instruments are typically used to generate paired-end reads as they hold significance for biological analysis. Owing to this treatment of our algorithm for paired-end reads, it is able to significantly improve the compression efficiency over state-of-the-art. More specifically, for a benchmark human dataset, the size of the compressed output is 21% smaller than that produced by SPRING, the current best algorithm. Further, ParRefCom is scalable and its compression and decompression speeds are better than or on par with those of *reference-free* methods.

CHAPTER 1
INTRODUCTION

Genome of an organism consists of one or more long DNA sequences called chromosomes, each a sequence of bases which are denoted by the four character alphabet {A, C, G, T}. Depending on the organism, the length of the genome can vary from several thousand bases to several billion bases. Genome sequencing, which involves deciphering the sequence of bases of the genome, is an important tool in genomics research. Sequencing instruments widely deployed today can only read short DNA sequences. However, these instruments can read up to several billion such sequences at a time, and are used to sequence a large number of randomly generated short fragments from the genome. These fragments are a few hundred bases long and are commonly referred to as *reads*. Each base in the genome is typically spanned by multiple reads. This oversampling is required to facilitate subsequent analysis of the reads. *Coverage* is defined as the average number of reads spanning a base in the genome.

Since the advent of short read high-throughput sequencing (HTS) machines, the cost of sequencing has been declining and the throughput has been increasing exponentially [2]. For instance, using the recent NovaSeq line of instruments from Illumina, the current market leader, sequencing cost is expected to come down to $100 per human genome. As a consequence of these revolutionary advances, HTS has found its use in a diverse range of applications beyond whole-genome sequencing. These include, among many others, genome resequencing, characterizing the transcriptome, cataloging of metagenomic samples, and providing personalized molecular diagnosis. See [3] and [2] for reviews of HTS and its applications.

The objective of this thesis is to develop parallel algorithms for enabling fast and scalable analysis of large-scale HTS short read datasets. Specifically, the thesis addresses the

following three problems related to HTS read datasets : (1) error correction, (2) partitioning, and (3) compression. In the remainder of this chapter, we briefly introduce each of these problems. A more detailed introduction, our proposed solution, and obtained results for these problems are described in Chapters 2, 3, and 4, respectively. Finally, our conclusions are provided in Chapter 5.

The work covered in this dissertation is published in papers [4] and [5]. The solutions presented in this dissertation are available as open-source software and can be accessed using the following links: `alurulab.cc.gatech.edu/parallel-ec` and `github.com/ParBLiSS/read_partitioning`.

1.1 Error Correction

Owing to limitations of sequencing technology, reads contain errors whose type is instrumentation-dependent. While the predominant error type in case of Illumina sequencing instruments is substitutions, it is insertions/deletions in case of Ion Torrent/Proton and Roche 454/GS [6, 7]. As Illumina sequencers are widely used today, we focus on correcting substitution errors. Read error correction is extremely important for two reasons. First, it improves the quality of results produced by downstream applications in areas such as genomics, metagenomics, and transcriptomics. Second, error correction leads to reduction in runtime and memory usage of such applications [8].

Illumina sequencers have low error rates, typically less than 1%. Combining this characteristic with the observation that each base in the genome is oversampled, many methods have been proposed to correct read errors. Spectrum-based Error Correction (SbEC) is a popular paradigm to correct substitution errors and it comprises of two major phases. The first phase is to construct a *kmer-spectrum* from the set of reads. In the second phase, read errors are detected and corrected by querying the kmer-spectrum. Owing to the large size of read datasets and the computationally intensive nature of error correction, there is an acute need for parallel error correction algorithms. Shah *et al.* proposed a distributed-memory

parallel algorithm for SbEC [1].

While the parallel algorithm presented by Shah *et al.* serves as a good starting point, it suffers from the following major drawbacks. First, a separate copy of the kmer-spectrum is maintained per MPI process in order to accomplish error correction. This approach does not scale to billion base long genomes (e.g. humans). Second, the work is statically allocated to processes during the error-correction phase. Because of differences in the distribution of errors among reads, this leads to significant load imbalance among processes. Third, error correction phase involves repeated binary searches over the kmer-spectrum. Binary search over a sorted layout is inefficient in terms of memory accesses given the organization of memory hierarchy in modern computer systems.

In this work, we address all of the above shortcomings to enhance the scope and the speedup of the parallel algorithm to accomplish read error correction of big genomic datasets. More specifically, we make the following contributions in order to achieve distributed-memory read error correction of a human dataset for the first time. First, we maintain only one copy of the kmer-spectrum per physical node instead of a copy of the kmer-spectrum per process. Second, we present a dynamic work allocation scheme to solve the load imbalance problem. Third, we propose a cache-aware layout to represent the kmer-spectrum to improve the memory-access efficiency. While the first optimization enables us to enhance the scope of error correction to big datasets, the second and the third allow us to improve the runtime substantially. We also present a parallel algorithm for constructing the cache-aware layout of the kmer-spectrum.

1.2 Read Partitioning

The large volumes of data generated from HTS experiments present significant computational challenges for downstream analysis pipelines. Analysis of such big datasets is compute intensive and requires a huge memory footprint. These challenges are expected to get more difficult, as more and more data becomes available in the future. Surveys note that

many laboratories spend significantly more time and resources on the computational analysis of HTS data than on generating the data itself [9, 10]. Many investigators attempt to address the memory demands by resorting to shared-memory machines with a large memory capacity. However, the number of hardware threads available for computation in such machines is limited. The challenges posed by ever growing sequencing throughput and the limitations suffered by currently available sequence analysis methods provide the context for our proposed solution.

In this work, we present an end-to-end distributed-memory parallel algorithm for partitioning HTS short read datasets. The goal of partitioning a large-scale HTS read dataset is to enable fast analyses using distributed-memory parallel systems. Our algorithm comprises of the following steps: (1) Building a de Bruijn graph from the given set of input reads, (2) Compacting the de Bruijn graph to reduce its size, (3) Partitioning the resulting compacted graph using a graph partitioner, and (4) Generating a partition of the read dataset from the partitioning of the de Bruijn graph. We also propose an algorithm to evaluate the quality of the partitioned read datasets.

We demonstrate that our solution produces high quality de Bruijn graph partitions, and as a consequence, high quality partitioning of read datasets, for an assortment of large-scale datasets. Such high quality partitioning enables fast distributed-memory parallel analyses of read datasets by keeping the communication necessary to execute the application low.

1.3 Read Compression

Transmission, storage, and archival of HTS short read datasets pose significant challenges owing to the large size of such datasets. Constant improvements to sequencing technology, in the form of increasing throughput and decreasing cost, and its growing adoption for a wide variety of applications amplify the problem. In response to this problem, researchers resorted to compression of read datasets.

General-purpose compression algorithms have been widely adopted for representing

HTS read datasets in a compact form. Read datatsets have several unique properties that make it difficult for general-purpose compressors to fully exploit the redundancy present in these datasets. Domain-specific properties of read datasets include reduced size of alphabet, interleaved streams of data, fixed length for reads, occurrence of reads and their reverse complements, paired representation of reads, scattered nature of redundancy, and availability of reference sequences. Researchers proposed special-purpose compression algorithms, that exploit one or more of these properties, to improve upon the compression efficiency of general-purpose compressors. Based on whether or not a reference sequence is made use of during compression, specialized compressors can be classified as reference-based or reference-free, respectively.

In this work, we leverage all of the above mentioned domain-specific properties of read datasets to develop ParRefCom, a parallel reference-based compressor for genomics read datasets. Read datasets generated using HTS instruments widely deployed today typically contain what are known as *paired-end* reads. A paired-end (PE) read is comprised of two separate but related reads, and the pairing information can serve to be crucial during biological analysis.

At a high-level, our solution approach consists of the following steps: (1) Specialized-alignment of PE reads to standard reference, (2) Categorizing PE reads based on the number of ends aligned, and (3) Customizing compression strategies for reads in different categories. In this work, we develop fast and scalable parallel algorithms for accomplishing each of these tasks. We demonstrate that ParRefCom achieves superior compression efficiency compared to existing methods. Our compressor is asymmetric by design - decompression speed is an order of magnitude faster than compression speed. This asymmetricity in design goes well with the real world requirement of compressing a dataset once and decompressing (using) it many times.

CHAPTER 2

PARALLEL READ ERROR CORRECTION FOR BIG GENOMICS DATASETS

2.1 Introduction

Genome of an organism consists of one or more long DNA sequences called chromosomes, each a sequence of bases which are denoted by the four character alphabet {A, C, G, T}. Depending on the organism, the length of the genome can vary from several thousand bases to several billion bases. Genome sequencing, which involves deciphering the sequence of bases of the genome, is an important tool in genomics research. Sequencing instruments widely deployed today can only read short DNA sequences. However, these instruments can read up to several billion such sequences at a time, and are used to sequence a large number of randomly generated short fragments from the genome. These fragments are a few hundred bases long and are commonly referred to as *reads*. Each base in the genome is typically spanned by multiple reads. This oversampling is required to facilitate subsequent analysis of the reads. *Coverage* is defined as the average number of reads spanning a base in the genome.

Owing to limitations of sequencing technology, reads contain errors whose type is instrumentation-dependent. While the predominant error type in case of Illumina sequencing instruments is substitutions, it is insertions/deletions in case of Ion Torrent/Proton and Roche 454/GS [6, 7]. As Illumina sequencers are widely used today, we focus on correcting substitution errors. Read error correction is extremely important for two reasons. First, it improves the quality of results produced by downstream applications in areas such as genomics, metagenomics, and transcriptomics. Second, error correction leads to reduction in runtime and memory usage of such applications [8].

The concepts described in the previous two paragraphs are illustrated in Figure 2.1.

Illumina sequencers have low error rates, typically less than 1%. Combining this characteristic with the observation that each base in the genome is oversampled, many methods have been proposed to correct read errors. Spectrum-based Error Correction (SbEC) is a popular paradigm to correct substitution errors and it comprises of two major phases. The first phase is to construct a *kmer-spectrum* from the set of reads. In the second phase, read errors are detected and corrected by querying the kmer-spectrum. Owing to the large size of read datasets and the computationally intensive nature of error correction, there is an acute need for parallel error correction algorithms. Shah *et al.* proposed a distributed-memory parallel algorithm for SbEC [1].

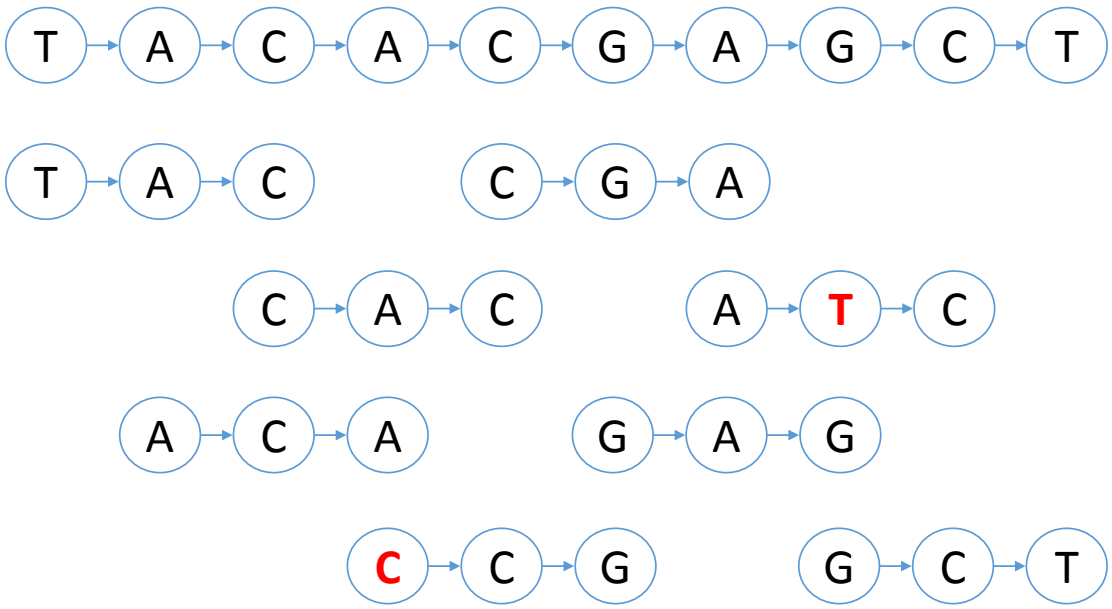

Figure 2.1: An illustration depicting various concepts related to read error correction. The long sequence represents the genome and the short sequences represent reads. Reads contain errors (highlighted) due to limitations of sequencing technology. The bases of the genome, and therefore read errors, are not known until after the reads are analyzed.

While the parallel algorithm presented by Shah *et al.* serves as a good starting point, it suffers from the following major drawbacks. First, a separate copy of the kmer-spectrum is maintained per MPI process in order to accomplish error correction. This approach does not scale to billion base long genomes (e.g. humans). Second, the work is statically allocated

to processes during the error-correction phase. Because of differences in the distribution of errors among reads, this leads to significant load imbalance among processes. Third, error correction phase involves repeated binary searches over the kmer-spectrum. Binary search over a sorted layout is inefficient in terms of memory accesses given the organization of memory hierarchy in modern computer systems.

In this work, we address all of the above shortcomings to enhance the scope and the speedup of the parallel algorithm to accomplish read error correction of big genomic datasets. More specifically, we make the following contributions in order to achieve distributed-memory read error correction of a human dataset for the first time. First, we maintain only one copy of the kmer-spectrum per physical node instead of a copy of the kmer-spectrum per process. Second, we present a dynamic work allocation scheme to solve the load imbalance problem. Third, we propose a cache-aware layout to represent the kmer-spectrum to improve the memory-access efficiency. While the first optimization enables us to enhance the scope of error correction to big datasets, the second and the third allow us to improve the runtime substantially. Finally, we present a parallel algorithm for constructing the cache-aware layout of the kmer-spectrum.

The rest of the chapter is organized as follows. To set up the stage for our work, we describe the working of the parallel SbEC algorithm proposed by Shah *et al.* in Section 2.2. Experimental methodology is explained in Section 2.3. Our solution strategies for addressing the three primary drawbacks of the parallel SbEC algorithm, along with the corresponding results and analyses, are presented in Sections 2.4, 2.5, and 2.6 respectively. Integrating the optimizations, we demonstrate error correction of a human dataset in Section 2.7. We furnish a survey of related work in Section 2.8.

2.2 Background

2.2.1 Spectrum-based Error Correction

Spectrum-based Error Correction (SbEC) methods are targeted towards correcting substitution errors in reads. Each read of length n is decomposed into $(n - k + 1)$ sub-reads called *kmers*, by reading substrings of length *k* starting at each position. Once the set of kmers is generated, each kmer is classified as either valid or invalid. Valid kmers are those that actually occur in the genome while invalid kmers are those that do not. The latter originate because of sequencing errors. As it is not possible to distinguish between valid and invalid kmers without knowledge of the genome, the following heuristic is used to categorize kmers. A kmer which occurs more frequently than a threshold value is labeled as valid, and is labeled invalid otherwise. This threshold value is influenced by the coverage to which the genome is sequenced. The objective of error correction is to convert invalid kmers into valid kmers with a minimum number of substitution operations. The correction choices for invalid kmers are generated by examining their valid neighbors using the Hamming distance metric.

2.2.2 Parallel Spectrum-based Error Correction

While the specific details of error correction vary among different spectrum-based algorithms, all of them are comprised of two important phases. The first phase involves constructing the kmer-spectrum, which represents the set of valid kmers. In the second phase, read errors are corrected by making use of the kmer-spectrum. Leveraging this observation, Shah *et al.* [1] proposed a generic SbEC framework for implementation on distributed-memory parallel computers. The parallel algorithm is composed of the following steps: (1) reads are evenly distributed among the processes, (2) each process generates an unordered multiset of kmers (represented as unsigned integers) based on the subset of reads allocated to it, (3) each process sorts its local list of kmers to eliminate copies and obtain frequencies,

(4) the entire list of kmers across all processes is globally sorted using a parallel sample sort, (5) a kmer-spectrum comprising of only valid kmers is built, (6) the kmer-spectrum is copied on all processes, and (7) error correction is performed independently on each process for the reads assigned to it. The implementation was demonstrated to scale with the number of processes, thus enabling fast error correction of datasets. Steps (1) – (6) make up the spectrum construction phase and step (7) corresponds to the error correction phase. The former phase is already well optimized. Moreover, the latter phase is the most time intensive step in the parallel algorithm and accounts for almost all the run time in case of big data sets. Therefore, in this work, we focus on enhancing the error correction phase of the algorithm.

2.2.3 Parallel Reptile

Shah *et al.* [1] used their parallel error correction framework to parallelize Reptile [11], an SbEC algorithm. We also use Parallel Reptile (PReptile) to demonstrate the benefits of our proposed optimizations. While the high level working of Reptile is similar to that of other spectrum-based algorithms, we describe one detail which sets up the context for interpreting the results of this work. Reptile constructs and makes use of two different spectra – kmer-spectrum and tile-spectrum. A tile is formed by concatenating two kmers and is of length $2k$. The use of the tile-spectrum, in addition to the kmer-spectrum, allows Reptile to better resolve ambiguities during error correction, thus enabling it to achieve better quality results. The use of PReptile, as a representative of spectrum-based methods for evaluation in this chapter, implies that searches are performed on two different spectra during the error correction phase.

2.3 Methodology

We ran our experiments on a cluster with a 40 Gbit QDR InfiniBand interconnect. Each node in the cluster has two 2.0 GHz 8-Core Intel Xeon E5 2650 processors, for a total of

16 cores per node. The 8 cores in a Xeon E5 2650 processor share a 20 MB last level cache and the 16 cores in a node share 128 GB of main memory. The cache line size on the Xeon E5 2650 processor is 64 bytes across all cache levels. We used this information while constructing cache-aware layouts for the kmer-spectrum and the tile-spectrum. The operating system is RedHat Enterprise Linux. For our experiments, we used up to 80 nodes in the cluster. Further, we used hybrid parallel programming with MPI and C++11 threads.

Table 2.1: Datasets used for experimental evaluation

Dataset	Organism	Read Count (millions)	Read Length (bases)	Coverage
D1	*H. sapiens*	1,550	101	50.5
D2	*E. coli*	8.9	101	193
D3A	*D. Melanogaster*	37.9	95	30
D3B	*Droso. M*	41.5	45	12
D3C	*Droso. M*	18.8	75	12

We used the same real datasets as Shah *et al.* [1] for evaluation, with the following exception. We replaced the original dataset D1, owing to its small size, with a human dataset. All the datasets are listed in Table 2.1 and are available from the NCBI Short Read Archive. Datasets D3A (SRX023452), D3B (SRX001651) and D3C (SRX001652) are combined into a single dataset D3, having 98.2 million reads. D2 (SRR034509_1) and D3 are used for our experiments in Sections 2.4 and 2.5. D1 (SRX027713 and SRX027583) is used for our experiments in Section 2.7. We used the same parameters as Shah *et al.* for correcting read errors and therefore, do not present results pertaining to quality of error correction.

2.4 Multiple Copies versus One Copy of Spectrum per Node

2.4.1 Motivation

In the Parallel Spectrum-based Error Correction (PSbEC) algorithm [1], Shah *et al.* maintain a copy of the spectrum per MPI process. For clusters which are prevalent in current times, multiple cores in a single node share a main memory. Even though the total capacity

of the main memory per node is fairy large, per core share of the main memory is typically small. As an example, in case of the cluster we used for running our experiments, 16 cores in a node share a 128 GB main memory. Therefore, the memory budget available per core or MPI process is only 8 GB. With such a small memory budget available per MPI process, it is not possible to perform error correction for big genomes and associated datasets. This is because the expected size of the spectrum is of the same order as the length of the genome. For the human dataset D1, the sizes of the kmer-spectrum and the tile-spectrum are 1,754,649,194 and 3,453,678,518 respectively. The per core memory budget of 8 GB is severely limited in comparison to the total amount of memory required to maintain both the spectra.

2.4.2 One Copy of Spectrum per Node

Our insight for addressing the first shortcoming of the PSbEC algorithm is as follows. Write operations are performed on the spectrum only during the spectrum construction phase. During the error correction phase, the only operations performed on the spectrum are reads. We can take advantage of this observation and maintain only one copy of the spectrum per node. In order to realize this in practice, we use hybrid parallelism with MPI and C++11 threads. More specifically, we launch an MPI job with as many processes as the number of nodes. Within a node, the MPI process launches as many threads as the number of cores. So, the total number of threads in our implementation is the same as the number of MPI processes in the PSbEC algorithm. During the error correction phase, only one copy of the spectrum is maintained per node or MPI process, and all the threads corresponding to an MPI process share and read from the same spectrum.

2.4.3 Results and Analysis

The primary objective of sharing the spectrum among the threads of a node or an MPI process is to enhance the scope of the PSbEC algorithm to accomplish error correction of big

datasets. By incorporating this optimization, in addition to realizing the primary goal, we are able to obtain performance improvement as a byproduct. The performance improvement obtained by transitioning from saving multiple copies of the spectrum to saving only one copy of the spectrum per node is shown in Tables 2.2 and 2.3 for datasets D2 and D3 respectively. Note that these results cannot be generated for the human dataset D1 as the original PSbEC algorithm cannot support the memory required by the dataset D1.

Table 2.2: Performance improvement obtained by replacing multiple copies of spectrum with one copy of spectrum per node for dataset D2 (Runtime is in seconds)

Multiple Copies		One Copy of Spectrum		
Processes	Runtime	Threads	Runtime	Improvement
128	2427	128	2273	6.77 %
256	1379	256	1347	2.38 %
512	708	512	713	-0.75 %
1024	365	1024	367	-0.47 %

Table 2.3: Performance improvement obtained by replacing multiple copies of spectrum with one copy of spectrum per node for dataset D3 (Runtime is in seconds)

Multiple Copies		One Copy of Spectrum		
Processes	Runtime	Threads	Runtime	Improvement
128	8983	128	8676	3.54 %
256	4986	256	5131	-2.83 %
512	3000	512	2829	6.05 %
1024	1844	1024	1844	-0.01 %

The results presented in Tables 2.2 and 2.3 demonstrate that the performance improvement obtained by storing only one copy of the spectrum per node can be as much as 7 $\%$. The source of this improvement is as follows. A core operation during the error correction phase is to lookup an element in the spectrum to determine its presence/absence. As the spectrum is represented as a sorted list of elements, the lookup operation comprises of performing a binary search in a sorted list. If the number of elements in the spectrum is N, the number of memory accesses made while performing the binary search is $O(log_2 N)$ in the worst case. Owing to temporal locality, some of these memory accesses correspond to cache hits. When we transition from storing multiple copies of the spectrum to storing only

one copy of the spectrum per node, more memory accesses are expected to correspond to cache hits. This is because the capacity of the last level cache, which is shared by multiple cores, is used towards storing elements of only one copy of the spectrum. In case of our cluster, 8 cores share a 20 MB last level cache.

While the maximum performance improvement is as much as 7 %, the improvement is not consistent as the total number of threads is increased from 128 to 1024. In some cases, the performance actually degrades. This is because of the imbalance in work performed across the threads. The same can be inferred from the runtime numbers in Tables 2.2 and 2.3, which do not scale with the number of processes/threads. Once the load imbalance across the threads is addressed, we can expect the performance improvement to remain consistent as we vary the number of threads. We tackle the second drawback of the PSbEC algorithm, pertaining to load imbalance across threads, in the following section.

2.5 Static versus Dynamic Work Allocation

2.5.1 Motivation

In the case of Parallel Spectrum-based Error Correction (PSbEC) algorithm proposed by Shah *et al.* [1], all the MPI processes are responsible for correcting the same number of reads. We refer to this scheme, where the reads are block decomposed among the MPI processes during the error correction phase, as the static work allocation mechanism. Conceptually, we expect that the work performed by each MPI process is roughly the same under the static allocation scheme. However, in practice, the static allocation scheme results in imbalance of work performed by various MPI processes. We ascertained that this is actually the case by measuring the time consumed during the error correction phase on individual MPI processes. This observed imbalance of work is due to the fact that the distribution of errors varies among reads. For a given number of reads, we further noticed that the imbalance becomes more pronounced as we increase the number of MPI processes. It must be noted that, in a parallel algorithm, the total runtime is determined by the time

taken by the longest running process. In order to solve the load imbalance problem experienced by the static work allocation scheme of the PSbEC algorithm, we propose a dynamic work allocation scheme. The latter scheme is based on a master-worker paradigm and is described next.

2.5.2 Dynamic Work Allocation Algorithm

As we specified in Sections 2.3 and 2.4, our overall solution uses hybrid parallel programming with MPI and C++11 threads. We create one MPI process per node and within a node, we create as many C++11 threads as there are cores. So, the total number of threads we create is the same as the total number of cores across all nodes. Our dynamic work allocation scheme uses a hierarchical master-worker paradigm and categorizes the threads into three different groups: global-master thread, local-master threads, and worker threads.

Preliminaries

On every node, one of the threads is classified as the local-master thread and the remaining as worker threads. So, there are as many local-master threads as there are nodes. Further, one thread among the local-master threads is selected as a global-master. Our master-worker model is set up hierarchically so that communication is carried out in the following manner. Local-master threads communicate with the global-master and vice-versa. Further, worker threads communicate only with the local-master corresponding to their node. In essence, there is only one communication channel emanating out of a node. We refer to a unit of work as a chunk and it comprises of a set of reads, whose size is parametrized. Further, a chunk is actually represented using two unsigned integers: (1) starting read ID and (2) chunk size. Each local-master maintains a work-queue of size $3 \times M$, where M represents the core count (also thread count) per node. The work-queue can hold up to $3 \times M$ chunks at any given time. We define a work-item as the number of chunks a local-master requests the global-master at a time. The size of the work-item in our implementation is

$2 \times M$. The tasks performed by various threads in our dynamic work allocation scheme are outlined in Figure 2.2 and described below.

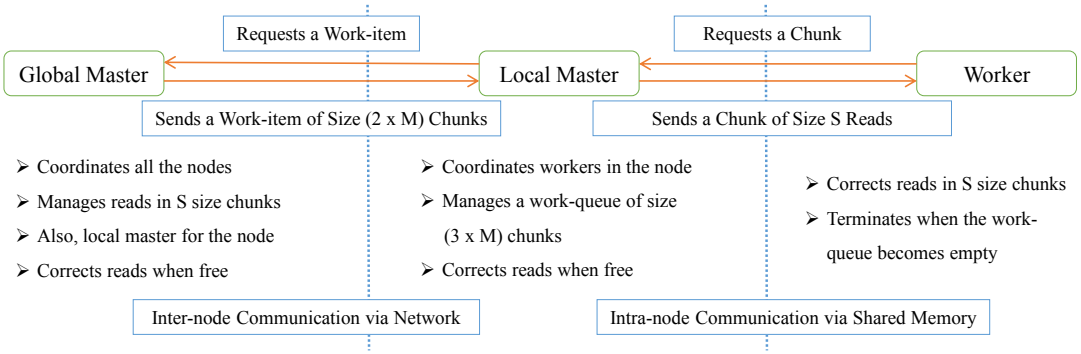

Figure 2.2: Roles played by global-master, local-master, and worker threads in a hierarchical master-worker paradigm for dynamic work distribution.

Global-master

The global-master thread is responsible for coordinating the entire dynamic work allocation process. The global-master has knowledge of the total number of reads that need to be corrected. It performs a logical partitioning of the entire read set into chunks of size S. A starting read ID and the chunk size S characterize a chunk. The global-master performs the following tasks. When a local master requests a work-item, it sends $2 \times M$ chunks to the local-master. A work-item represents a subset of reads that needs to be corrected next. If there are no more reads to be corrected, the global-master communicates the same information to a local-master when the latter requests a work-item. After communicating the $no - more - work$ message to all the local-masters, the global-master waits for all the local-masters to complete the work previously requested by them. The global-master, in addition to discharging the responsibilities just described, plays the role of a local-master for the worker threads in the same node.

Local-master

In our dynamic work allocation algorithm, we create one local-master thread per node. The local-master acts as a conduit between the global-master and worker threads of the same node. Each local-master is responsible for maintaining a work-queue of size $3 \times M$ chunks. Whenever the occupancy of the work-queue falls to M chunks, the corresponding local-master requests the global-master for a work-item of size $2 \times M$ chunks. Once the local-master receives the requested work-item, it pushes $2 \times M$ chunks into the work-queue. The work-queue and the work-item sizes are designed such that the worker threads never starve for work. We will elaborate more on this aspect below. The local-master continues to request for work-items until it receives a $no - more - work$ message form the global-master. Once the local-master receives the $no - more - work$ message from the global-master, the former waits for the work corresponding to the chunks currently in the work-queue to be completed. The local-master then sends a $pending - work - done$ message to the global-master and exits. The local-master, in addition to discharging the responsibilities just described, corrects reads when it is free.

Worker

The role played by a worker thread is relatively straightforward. A worker-thread fetches a chunk from the work-queue and performs error correction for the subset of reads corresponding to the chunk. When the worker thread is done with a chunk of reads, it fetches another chunk from the work-queue. This process is repeated until the work-queue becomes empty and the local-master receives the $no-more-work$ message from the global-master. When the worker-thread reaches this point during execution, it informs the local-master and exits.

Chunk size

Given our choice of the sizes of the work-queue and the work-item, the only parameter in the dynamic work allocation algorithm is chunk size. Now, we provide insight into choosing the right value for this parameter. The chunk size must be big enough to hide the round-trip latency between a local-master and the global-master. This is because there are only M chunks in the work-queue, one chunk per worker thread on average, when the local-master requests a work-item. Therefore, the time required to correct the subset of reads making up a chunk must be long enough to cover the round-trip latency and any delay at the global-master in servicing the request. On the other hand, the chunk size must be small enough so that the time difference among nodes while draining out the queue after receiving the $no-more-work$ message from the global-master is insignificant. By taking the network-latency of our cluster into account, a choice of 500 reads for the chunk size achieves a good trade-off between the two conflicting requirements.

Note that in our implementation, the global-master and the local-masters request work for themselves. This allows them to perform useful work when they are not discharging their administrative responsibilities. All the worker threads exit before the corresponding local-master exits and all the local-masters exit before the global-master exits. This ensures smooth termination of the overall algorithm. When the error correction phase commences, we initialize the queues corresponding to all local-masters using a block decomposition of reads into chunks. This avoids the situation of all local-masters requesting the global-master for work concurrently during initialization. The starting read ID on the global-master is set after taking this initial allocation into consideration. Therefore, the global-master can generate subsequent work-items correctly.

2.5.3 Results and Analysis

The speedup obtained by replacing the static work allocation scheme of the PSbEC algorithm with our dynamic work allocation scheme is shown in Tables 2.4 and 2.5 for datasets

D2 and D3 respectively. The results under the 'Static Runtime' and the 'Dynamic Runtime' columns correspond to the static and the dynamic work allocation schemes respectively. It can be noticed that a maximum speedup of 6.53X is obtained in case of dataset D3. In general, across both datasets D2 and D3, the speedup increases as the number of threads is increased. This trend is in accordance with our observation in Section 2.4.3 that the load imbalance increases with the number of threads. More importantly, the numbers under the 'Dynamic Runtime' column scale in a near perfect manner as the number of threads is increased. This trend corroborates the value of our dynamic work allocation scheme.

Table 2.4: Performance improvement obtained by replacing static work allocation with dynamic work allocation for dataset D2 (Runtime is in seconds)

Threads	Static Runtime	Dynamic Runtime	Speedup
128	2273	1210	1.88
256	1347	618	2.18
512	713	320	2.23
1024	367	167	2.20

Table 2.5: Performance improvement obtained by replacing static work allocation with dynamic work allocation for dataset D3 (Runtime is in seconds)

Threads	Static Runtime	Dynamic Runtime	Speedup
128	8676	2231	3.89
256	5131	1117	4.59
512	2829	561	5.05
1024	1844	282	6.53

In our experiments, we used as many as 80 nodes of the cluster. Even for the most challenging configuration involving 80 nodes, the global-master in our hierarchical master-worker scheme has spare cycles available after discharging the administrative responsibilities assigned to it. Owing to this reason, the global-master is able to correct chunks of reads.

2.6 Memory-access Efficient Data Layout

2.6.1 Motivation

As mentioned in Section 2.2.2, the error correction phase is the most time consuming step of the PSbEC algorithm and accounts for almost all the runtime in case of big datasets. A core operation in this phase involves looking up elements in the spectrum. Since the spectrum is stored as a sorted list, look up operations involve repeated binary searches over the spectrum. The memory subsystem in modern computers is organized as a hierarchy, with multiple levels of cache. Further, data is transferred across the hierarchy at cache line size granularity. Binary search over a sorted list is inefficient in terms of memory accesses given the memory organization of modern computer systems. This is because binary search does not take advantage of the spatial locality property, which is a key factor in extracting the maximum performance from the memory subsystem. In order to address this deficiency, we propose to store the spectrum as a cache-aware layout.

2.6.2 Cache-aware Layout

We use the term 'default layout' to refer to the spectrum stored as a sorted list. The spectrum is stored as a default layout in case of the PSbEC algorithm, which serves as the baseline. Further, we use N to represent the total number of elements in the spectrum and B to represent the number of elements that can fit into one cache line. Binary search in the default layout incurs $O(log_2 N)$ lookups when the search is unsuccessful. This cost can be reduced to $O(log_{(B+1)} N)$ lookups by arranging the elements using a cache-aware layout. The cache-aware layout relies on the knowledge of the size of the cache line and hence the name. Note that the cache-aware layout we use in this work is static in nature, in that it is not updated once constructed.

The relationship between a cache-aware layout and its cache-aware search tree is analogous to that between a default layout and its binary search tree. A cache-aware tree is

organized as a $(B+1)$-ary tree, with each node in the tree having $(B+1)$ children in a complete tree. An example cache-aware search tree and its binary counterpart are shown in Figure 2.4 and 2.3 respectively for N=26 and B=2. The height of a cache-aware tree is $log_{(B+1)}(N+1)$. An unsuccessful search in such a tree incurs $O(log_{(B+1)}N)$ memory accesses. This is a factor of $O(log_2(B+1))$ improvement when compared to an unsuccessful search in the corresponding binary tree. A node in a cache-aware tree consists of B elements, which among them define $(B+1)$ ranges. $(B+1)$ subtrees of this node are arranged such that the elements in a subtree fall in the corresponding range. In Figure 2.4, the node containing elements $\{9, 18\}$ has three subtrees. The elements in these subtrees are < 9, > 9 & < 18, and > 18, respectively.

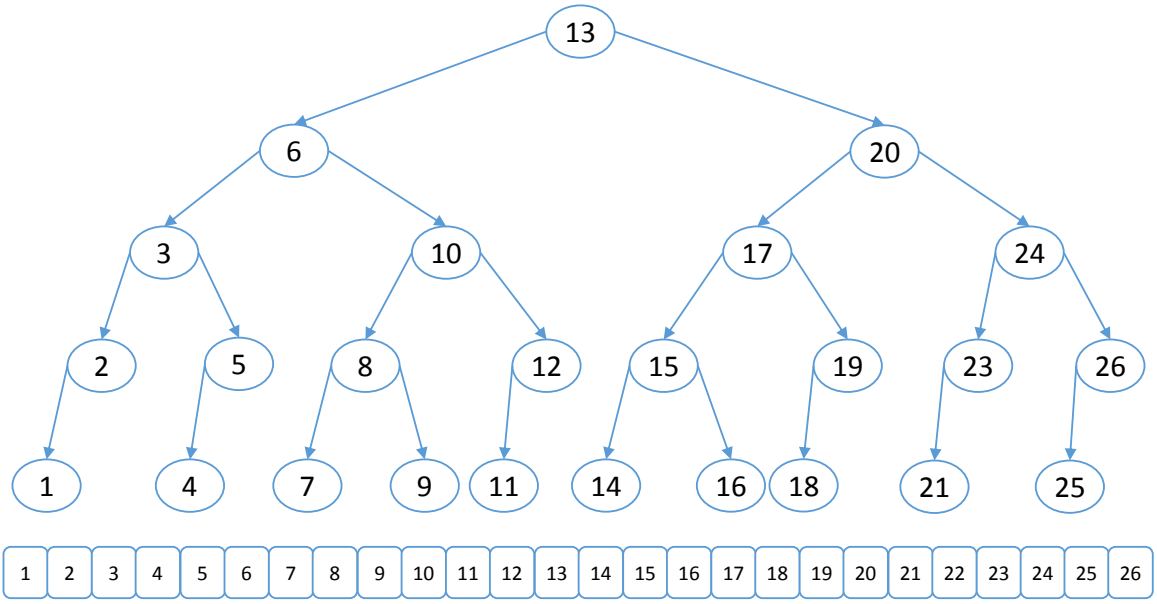

Figure 2.3: Binary search tree for a sorted list of 26 elements with contents $[1, 2, \ldots, 26]$ in case of default layout.

Construction

The cache-aware layout corresponding to a cache-aware tree is obtained by laying out the nodes in level order from root level to leaf level. Within a level, the nodes are laid out from left to right. Our algorithm for constructing the cache-aware layout is presented in

Algorithm 1: Cache-aware Layout Construction

$Build - Cache - Aware - Layout(CAL, B, X, S, N)$
Input : B, Elements per cache line ($B \geq 3$).
Input : X, Default layout (also sorted list).
Input : S, Start index for processing.
Input : N, No. of elements to process. // $N \bmod B$ is 0.
Output : CAL, Cache-aware layout of X.
$L \leftarrow \lceil \log_{B+1}(N+1) \rceil$ // Global variable.
$A_1[i] = B(B+1)^i, 0 \leq i < L$ // Global array.
$A_2[i] = \sum_{j=0}^{i} A_1[j], 0 \leq i < L$ // Global array.
$i \leftarrow 0; CAL[S] \leftarrow S; CAL[S+1] \leftarrow L; CAL[S+2] \leftarrow S + N - 1$
$c_{idx} \leftarrow S; n_{idx} \leftarrow S$ // Current index and next index.
while $i < N$ **do**
    // $[x, y]$: Range covered by current subtree.
    $x \leftarrow CAL[c_{idx}]; \quad y \leftarrow CAL[c_{idx} + 2]$
    // l: No. of levels in current subtree.
    $l \leftarrow CAL[c_{idx} + 1]; \quad d \leftarrow y - x + 1$
    $ST \leftarrow Subtree - Size(d, l)$
    $z \leftarrow x$ // Current node.
    for $j \leftarrow 0$**to**$(B-1)$ **do**
        $z \leftarrow z + ST[j]; CAL[c_{idx} + j] \leftarrow X[z]; z = z + 1$
    end
    $z \leftarrow x$ // Subtree roots.
    for $j \leftarrow 0$**to**B **do**
        if $ST[j] > 0$ **then**
            $n_{idx} \leftarrow n_{idx} + B; CAL[n_{idx}] \leftarrow z; CAL[n_{idx} + 1] \leftarrow l - 1$
            $z \leftarrow z + ST[j]; CAL[n_{idx} + 2] \leftarrow z - 1; z \leftarrow z + 1$
        end
    end
    $i = i + B; c_{idx} = c_{idx} + B$
end

$Subtree - Size(ST, d, l)$
Input : d, Tree size; l, Tree levels.
Output : ST, $B+1$ subtree sizes.
if $d == B$ **then**
    return ST with $(B+1)$ zeros. // Last row : All zeros.
end
$d_l \leftarrow d - A_2[l-2]; q = d_l / A_1[l-2]; r = d_l \bmod A_1[l-2]$
$ST[q] \leftarrow A_2[l-2]$
for $j \leftarrow 0$**to**$(q-1)$ **do**
    $ST[j] \leftarrow A_1[l-2] + A_2[l-2]$
end
for $j \leftarrow (q+1)$**to**B **do**
    $ST[j] \leftarrow r + A_2[l-2]$
end

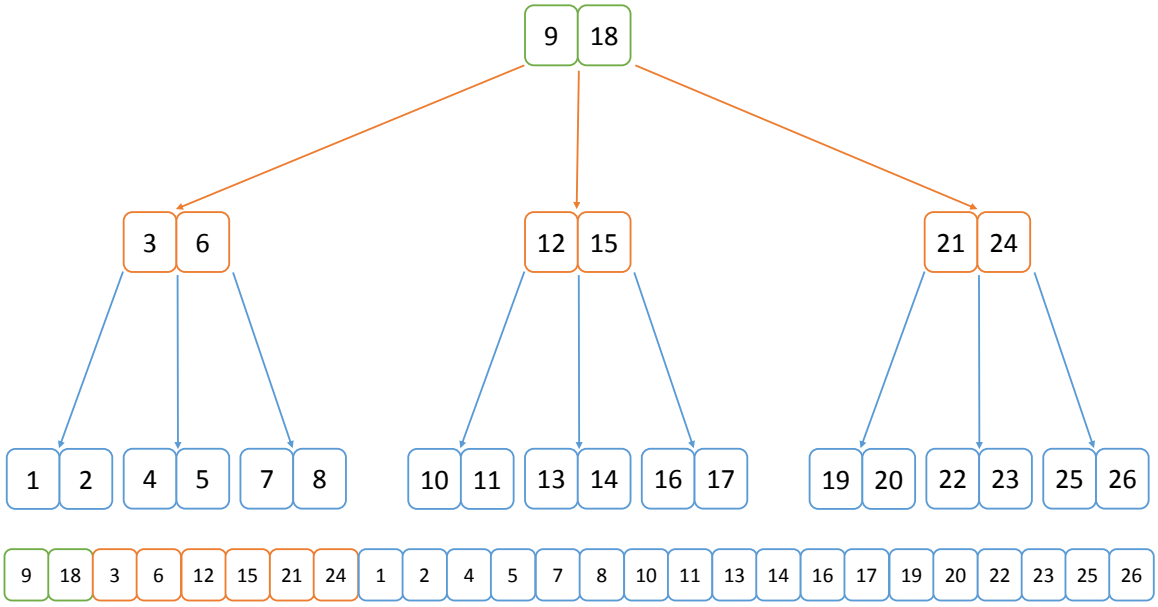

Figure 2.4: Cache-aware search tree for a sorted list of 26 elements with contents $[1, 2, \ldots, 26]$ in case of cache-aware layout.

Algorithm 1. While processing a node x, the algorithm computes the size of each of its $B + 1$ subtrees, and thus determines the B elements of node x. The start and end indices of each of the $B + 1$ subtrees are calculated and temporarily stored in the corresponding children of x. The algorithm then proceeds to process the child nodes of x. The algorithm works on all nodes at a particular level before proceeding to work on a level below it.

Lookup

The algorithm for looking up an element in the cache-aware layout is presented in Algorithm 2. The search process commences at the root node of the tree and progresses down the tree one level at a time. When the element being looked up is not found in the tree, the search falls off of a leaf node of the tree. At a particular node, the algorithm parses up to B elements of the node to determine the appropriate branch to pursue. If the element being looked up is found in a node, the search process terminates and reports a success. The maximum number of memory accesses required to look up an element corresponds to the case when the element is not present in the cache-aware layout, and is $O(log_{(B+1)}N)$.

Algorithm 2: Lookup Element in Cache-aware Layout

```
1  Search − In − Cache − Aware − Layout(B, CAL, E)
2  Input : B, Elements per cache line.
3  Input : CAL, Cache-aware layout. N ← |CAL|.
4  Input : E, Element to be looked up in CAL.
5  Output : If E is found in CAL, TRUE. Else, FALSE.
6  L ← ⌈log_{B+1}(N + 1)⌉ // Global variable.
7  A_1[i] = B(B + 1)^i, 0 ≤ i < L // Global array.
8  A_2[i] = Σ_{j=0}^{i} A_1[j], 0 ≤ i < L // Global array.
9  c_idx ← 0 // Current index.
10 i ← j_curr ← j_prev ← 0 // Local variables.
11 for i ← 0to(L − 1) do
12 |   for j_curr ← 0to(B − 1) do
13 |   |   if (CAL[c_idx + j_curr] == E) then
14 |   |   |   return TRUE
15 |   |   end
16 |   |   if (CAL[c_idx + j_curr] > E) then
17 |   |   |   last
18 |   |   end
19 |   end
20 |   c_idx ← A_2[i] + j_prev × A_1[i] + j_curr × B
21 |   j_prev ← j_curr
22 end
23 return FALSE
```

2.6.3 Parallel Construction and Lookup

Parallel Construction of Cache-aware Layout

The algorithms we presented in Section 2.6.2 are useful for construction of and lookup in a cache-aware layout using a single thread. However, for clusters which are prevalent in current times, multiple cores in a single node share the main memory. We can construct the cache-aware layout in the fastest possible time if we can leverage the compute capability of all the cores. Such an algorithm for constructing the cache-aware layout in parallel is presented in Algorithm 3. We carefully designed this algorithm such that it builds on top of Algorithm 1 and has negligible serial overhead. The sequential portion of the algorithm just comprises of selecting T (number of cores or threads) elements, which form the

starting elements for T equal-sized logical partitions. Each thread is then responsible for constructing the cache-aware layout of a logical partition.

Algorithm 3: Parallel Cache-aware Layout Construction

```
1  Par − Build − Cache − Aware − Layout(CAL, SLP, T, B, X)
2  Input : T, Number of threads.
3  Input : B, Elements per cache line (B ≥ 3).
4  Input : X, Default layout (also sorted list). N ← |X|.
5  Output : CAL, Cache-aware layout of X.
6  Output : SLP, Start elements of logical partitions.
   // N mod (B × T) is 0.
   // Partition X into T logical parts.
   // Each thread will handle one logical partition.
7  t ← 0 // Local variable.
8  for t ← 0to(T − 1) do
9  |  SLP[t] ← X[t×N/T]
10 end
11 parallel for t ← 0to(T − 1) do
12 |  q ← t×N/T // Start index for processing.
13 |  r ← N/T // No. of elements to process.
   |  // CAL − Thread function below is the same as
   |  // Build − Cache − Aware − Layout function
   |  // in Algorithm 1.
14 |  CAL − Thread(CAL, B, X, q, r)
15 end
16 return (CAL, SLP)
```

Lookup in Parallel Cache-aware Layout

Our algorithm for looking up an element in a cache-aware layout, which is constructed in parallel, is presented in Algorithm 4. This algorithm is also designed such that it builds on top of Algorithm 2. More specifically, the functionality of the block in lines 21-32 of Algorithm 4 is the same as that of the block in lines 11-22 of Algorithm 2. However, owing to minor differences in the construction of sequential and parallel cache-aware layouts, the variables c_{idx} and L are initialized differently. The algorithm starts off by determining the logical partition to which the element being looked up belongs to. After this, the search process works as in Algorithm 2, on the identified logical partition, to report a success or a

failure. The maximum number of memory accesses required to look up an element in case of Algorithm 4 is also $O(log_{(B+1)}N)$.

Algorithm 4: Lookup in Parallel Cache-aware Layout

$Search - In - Parallel - CAL(T, B, CAL, SLP, E)$
Input : T, Number of threads.
Input : B, Elements per cache line.
Input : CAL, Cache-aware layout. $N \leftarrow |CAL|$.
Input : SLP, Start elements of logical partitions.
Input : E, Element to be looked up in CAL.
Output : If E is found in CAL, TRUE. Else, FALSE.
$L \leftarrow \lceil \log_{B+1}(\frac{N}{T} + 1) \rceil$ `// Global variable.`
$A_1[i] = B(B+1)^i, 0 \leq i < L$ `// Global array.`
$A_2[i] = \sum_{j=0}^{i} A_1[j], 0 \leq i < L$ `// Global array.`
$i \leftarrow t \leftarrow j_{curr} \leftarrow j_{prev} \leftarrow 0$ `// Local variables.`
for $t \leftarrow 0$**to**$(T-1)$ **do**
    if $(SLP[t] > E)$ **then**
        $last$
    end
end
if $(t == 0)$ **then**
    return $FALSE$
end
$c_{idx} \leftarrow \frac{(t-1) \times N}{T}$ `// Current index.`
for $i \leftarrow 0$**to**$(L-1)$ **do**
    for $j_{curr} \leftarrow 0$**to**$(B-1)$ **do**
        if $(CAL[c_{idx} + j_{curr}] == E)$ **then**
            return $TRUE$
        end
        if $(CAL[c_{idx} + j_{curr}] > E)$ **then**
            $last$
        end
    end
    $c_{idx} \leftarrow A_2[i] + j_{prev} \times A_1[i] + j_{curr} \times B$
    $j_{prev} \leftarrow j_{curr}$
end
return $FALSE$

2.6.4 Results

The speedup obtained by substituting default layout with cache-aware layout is shown in Tables 2.6 and 2.7 for datasets D2 and D3 respectively. In our implementation, kmers and tiles are represented using 4 byte and 8 byte unsigned integers respectively. Further, the cache line size is 64 bytes. Therefore, the branching factor of the cache-aware tree is 17 ($\leftarrow \frac{64}{4} + 1$) and 9 ($\leftarrow \frac{64}{8} + 1$) for kmer and tile spectra respectively. For both datasets, the maximum speedup is 1.7 X. Further, the speedup remains almost constant as the number of threads is varied. This is because our dynamic work allocation algorithm is able to distribute the load evenly across the threads, independent of the layout of the spectra.

The runtime under 'Cache-aware Time' column in Tables 2.6 and 2.7 includes the cache-aware layout construction time. In order to correct invalid kmers/tiles of erroneous reads, queries are placed in search of valid kmers/tiles by enumerating all possible kmers/tiles at a hamming distance of up to 2 ($h \leftarrow 2$). We chose this value as it was previously shown that using $h \leftarrow 2$ leads to high quality error correction [1].

Table 2.6: Performance improvement by replacing default layout with cache-aware layout for dataset D2 (Time is in seconds)

Threads	Default Time	Cache-aware Time	Speedup
128	1210	725	1.67
256	618	367	1.68
512	320	189	1.69
1024	167	98	1.70

Table 2.7: Performance improvement by replacing default layout with cache-aware layout for dataset D3 (Time is in seconds)

Threads	Default Time	Cache-aware Time	Speedup
128	2231	1314	1.70
256	1117	659	1.70
512	561	332	1.69
1024	282	168	1.68

2.6.5 Analysis

We also evaluated cache-oblivious layout [12, 13] in order to address the memory-access inefficiency associated with the default layout. The relationship between a cache-oblivious layout and its cache-oblivious search tree is analogous to that between a default layout and its binary search tree. Cache-oblivious data structures and algorithms target optimum number of memory transfers without the knowledge of organization and parameters of the memory subsystem [12]. The algorithm for constructing a cache-oblivious tree from a binary tree works as follows. The binary tree is partitioned at half of its height. This process results in a top subtree and $O(\sqrt{N})$ bottom subtrees, each of size $O(\sqrt{N})$. The partitioning process continues recursively on each subtree until we end up with subtrees which can fit into one or two cache lines. It is common to layout subtrees in level order starting from the topmost level, although this is not a requirement. The subtrees within a level are laid out in left to right order. The algorithms for constructing and accessing the cache-oblivious layout are described in detail by Ronn [13]. An unsuccessful search in the cache-oblivious tree can incur up to $4 \times log_B N$ accesses in the worst case. The speedup obtained by substituting default layout with cache-oblivious layout is shown in Tables 2.8 and 2.9 for datasets D2 and D3 respectively. As the cache-aware layout is able to leverage the knowledge of the cache line size, it performed consistently better than the cache-oblivious layout in our evaluation.

Table 2.8: Performance improvement by replacing default layout with cache-oblivious layout for dataset D2 (Time is in seconds)

Threads	Default Time	Cache-oblivious Time	Speedup
128	1210	877	1.38
256	618	441	1.40
512	320	230	1.39
1024	167	118	1.41

Now, we analyze the reasons for the discrepancy between the expected and the actual performance of cache-aware and cache-oblivious layouts. Many of them are common be-

Table 2.9: Performance improvement by replacing default layout with cache-oblivious layout for dataset D3 (Time is in seconds)

Threads	Default Time	Cache-oblivious Time	Speedup
128	2231	1594	1.40
256	1117	798	1.40
512	561	406	1.38
1024	282	203	1.39

tween the two layouts and are as follows. First, cache-aware search tree exploits spatial locality for performance. The key insight is to bring the elements, which are accessed consecutively in a binary search tree, together in a spatial sense. However, as the topmost levels of the binary search tree are accessed repeatedly, they benefit from temporal locality. We expect that the elements in the top 20 levels of the binary tree experience cache hits due to temporal locality. The space required by the top 20 levels of the binary tree for kmers and tiles is 4 MB and 8 MB respectively. The total space requirement is 12 MB, whereas the capacity of the last level cache is 20 MB. The second factor is related to the characteristic that not all lookups will be unsuccessful, and those lookups that are successful can hit at various levels in the tree. Finally, elements in the bottom $log_2 B$ levels of the binary tree benefit from spatial locality as well. The three factors outlined above account for the difference between the expected and the actual performance of memory-access efficient layouts.

In addition, cache-oblivious layout suffers from two disadvantages, which are unique to it. First, some of the lowest level subtrees in cache-oblivious layout can suffer from bad alignment resulting in them staggering across two cache lines. Second, the lack of knowledge of the cache line size affects this layout. These two factors are primarily responsible for the slower speed up of cache-oblivious layout when compared to cache-aware layout. However, incomplete cache-oblivious search trees experience a slight advantage in our implementation due to the manner in which we handle them. Specifically, we handle an incomplete cache-oblivious search tree as a collection of multiple complete cache-oblivious search trees of successively smaller sizes, to manage complexity.

2.7 Supporting Big Datasets

In Sections 2.4, 2.5, and 2.6, we developed strategies to address three major shortcomings of the Parallel Spectrum-based Error Correction (PSbEC) algorithm. As our first optimization, we maintained only one copy of the spectrum per physical node instead of a copy of the spectrum per process. Second, we designed a dynamic work allocation scheme to solve the load imbalance problem. Third, we proposed to represent the spectrum using a cache-aware layout in order to improve the memory-access efficiency. Combining these three optimizations allows us to expand the scope of the PSbEC algorithm to accomplish error correction of big genomic datasets. Such datasets arise in case of long and complex genomes.

As a demonstration of the capability of our enhanced PSbEC algorithm, we performed error correction of a human dataset (D1) consisting of nearly 1.55 billion reads. Other details related to this dataset are available in Table 2.1. For the human dataset D1, the sizes of the kmer-spectrum and the tile-spectrum are 1,754,649,194 and 3,453,678,518 respectively. For the original PSbEC algorithm, the per core memory budget of 8 GB on our cluster is severely limited in comparison to the total amount of memory required to maintain both the spectra. Using our enhanced PSbEC algorithm, we are able to complete error correction of dataset D1 in 8.7 hours using 1280 threads ($\leftarrow 80$ nodes $\times$ 16 $\frac{cores}{node}$).

Dataset D1 cannot be corrected using the original PSbEC algorithm on our cluster as the memory available is significantly less than the memory required. Further, the cumulative speedup due to our proposed optimizations is 11 X for dataset D3, when 1024 threads are used. Based on this information, we believe that error correction using the original PSbEC algorithm would be much slower in comparison to our enhanced PSbEC algorithm under the hypothetical condition that memory is not a constraint. These results corroborate the importance of our enhanced PSbEC algorithm for performing error correction of big datasets. Note that the maximum number of nodes we are able to use on the cluster is 80.

2.8 Related Work

Genomic read error correction received a lot of attention over the years due to its prominence. The error correction utilities can be mainly classified into three categories: (1) Spectrum based, (2) Suffix array/tree based, and (3) Multiple Sequence Alignment (MSA) based [14]. While the spectrum-based algorithms are meant for correcting substitution errors, algorithms in the other two categories can also correct insertion/deletion errors. Due to the predominant use of Illumina sequencers, which mainly make substitution errors, spectrum-based algorithms received the most attention. As our work also falls into this class, we only survey prior work in the spectrum-based algorithms category. For information related to tools in other categories, we refer the reader to the survey by Yang *et al.* [14].

There is a vast body of work in the spectrum-based algorithms category meant for correcting substitution errors [15]. The Parallel Spectrum-based Error Correction (PSbEC) algorithm proposed by Shah *et al.* [1], and therefore our Enhanced Parallel Spectrum-based Error Correction (EPSbEC) algorithm, provides a generic framework to parallelize any SbEC algorithm. Two characteristics are key to developing a parallel SbEC algorithm: (1) a way to construct the spectrum in parallel, and (2) a means for supporting queries to the spectrum in parallel. The PSbEC and EPSbEC algorithms support both of these aspects. Reptile [11] is just used as a representative of the category of SbEC algorithms to demonstrate the parallelization strategies. Given this context, we refrain from discussing any SbEC algorithm specifically. Instead, we refer to the work by Molnar *et al.* [15] for an extensive survey of various SbEC algorithms.

In recent related work, Molnar *et al.* [15] evaluated the performance (both biological and computational) of various SbEC tools on a human dataset. While most of the tools failed to handle the human dataset due to its size, three tools – Musket [16], RACER [17], and SGA [18] – ran to completion. All three tools use shared-memory parallelism and

were run on a shared-memory machine with 32 cores and 1024 GB of memory. RACER and SGA, which produced results with the best biological quality, required a large amount of memory and/or incurred a long runtime. As our EPSbEC algorithm is designed for distributed-memory systems and scales almost perfectly with the number of nodes, it addresses the limitations of the algorithms designed for shared-memory in order to realize significant performance benefits.

DecGPU [8] and PSbEC [1] are the only SbEC algorithms designed to run on distributed-memory systems. DecGPU was demonstrated only for 32 cores and did not show good parallel scalability. In this work, we enhanced the PSbEC algorithm to improve its scope and speedup significantly.

The cache-aware layout described in Section 2.6 is similar in spirit to a B-tree [19]. However, in contrast to a B-tree, our cache-aware layout of the spectrum is static in nature in that it is not updated once constructed. As our cache-aware layout does not need to support insertion/deletion of elements, it is more efficient in comparison to a general B-tree.

CHAPTER 3

DISTRIBUTED MEMORY PARTITIONING OF HTS READ DATASETS

3.1 Introduction

Since the advent of short read high-throughput sequencing (HTS) machines, the cost of sequencing has been declining and the throughput has been increasing exponentially [2]. For instance, using the recent NovaSeq line of instruments from Illumina, the current market leader, sequencing cost is expected to come down to $100 per human genome. As a consequence of these revolutionary advances, HTS has found its use in a diverse range of applications beyond whole-genome sequencing. These include, among many others, genome resequencing, characterizing the transcriptome, cataloging of metagenomic samples, and providing personalized molecular diagnosis. See [3] and [2] for reviews on HTS and its applications.

Unfortunately, the large volumes of data generated from HTS experiments present significant computational challenges for downstream analysis pipelines. Analysis of such big datasets is compute intensive and requires a huge memory footprint. These challenges are expected to get more difficult, as more and more data becomes available in the future. Surveys note that many laboratories spend significantly more time and resources on the computational analysis of HTS data than on generating the data itself [9, 10]. Many investigators attempt to address the memory demands by resorting to shared-memory machines with a large memory capacity. However, the number of hardware threads available for computation in such machines is limited. The challenges posed by ever growing sequencing throughput and the limitations suffered by currently available sequence analysis methods provide the context for our proposed solution.

In this chapter, we present an end-to-end distributed-memory parallel algorithm for par-

titioning HTS short read datasets. The goal of partitioning a large-scale HTS read dataset is to enable fast analyses using distributed-memory parallel systems. Our algorithm comprises of the following steps: (1) Building a de Bruijn graph from the given set of input reads, (2) Compacting the de Bruijn graph to reduce its size, (3) Partitioning the resulting compacted graph using a graph partitioner, and (4) Generating a partition of the read dataset from the partitioning of the de Bruijn graph. We also propose an algorithm to evaluate the quality of the partitioned read datasets.

Excluding sequencing errors, the size of the de Bruijn graph is expected to track the cumulative length of the source DNA molecules from which the read dataset is generated. Every read traces a path in the de Bruijn graph and there are overlaps among paths traced by multiple reads. Based on these properties, our insight is that if we can generate a good partitioning of the de Bruijn graph associated with a set of reads, it can be translated into a good partitioning of the read set itself. We demonstrate that our solution produces high quality de Bruijn graph partitions, and as a consequence, high quality partitioning of read datasets, for an assortment of large-scale datasets. Such high quality partitioning enables fast distributed-memory parallel analyses of read datasets by keeping the communication necessary to execute the application low.

Since our approach leverages de Bruijn graph as an intermediate data structure, we are able to generate partitions of a set of reads without computing pairwise similarity among reads, which is known to be challenging both in terms of computation and memory. Variants of de Bruijn graph gained significant adoption for analyses of HTS reads, a prominent example being de novo genome assembly. State-of-the-art distributed-memory parallel assemblers use hashing to distribute the de Bruijn graph. This approach destroys data locality, and hence increases communication costs during downstream analyses. In contrast, our approach for partitioning the de Bruijn graph respects locality. Therefore, the partitioned de Bruijn graph we generate can be used by analyses that predominantly make use of some variant of the graph.

We present our proposed algorithms for de Bruijn graph and read set partitioning in Sections 3.4 and 3.5 respectively. Using real datasets, the respective sections also discuss the quality of the generated partitions and the corresponding runtime incurred. Information about datasets and our methodology is provided in Section 3.3. Before we proceed to describe the proposed algorithms, we review the notations and relevant data structures used in this chapter in Section 3.2.

3.2 Background

3.2.1 Representation of Sequences

A DNA molecule is comprised of two strands. The forward strand of the molecule, defined in the $5'$ to $3'$ direction, is modeled as a sequence of characters from the alphabet set $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Given such a sequence of n characters, $s = s_1, \ldots, s_n$, its reverse complementary strand *i.e.,* the sequence in the $3'$ to $5'$ direction, is $\overline{s} = c(s_n), \ldots, c(s_1)$, where $c(x), x \in \Sigma$ is the mapping function : $c(\mathtt{A}) \rightarrow \mathtt{T}, c(\mathtt{C}) \rightarrow \mathtt{G}, c(\mathtt{G}) \rightarrow \mathtt{C}$, and $c(\mathtt{T}) \rightarrow \mathtt{A}$. A DNA molecule can be represented by the pair $(s, \overline{s})$ or only by the *canonical sequence* $\hat{s}$, which is the lexicographically smaller of the two strings s and $\overline{s}$ *i.e.,* $\min(s, \overline{s})$. Unless specified otherwise, we use the canonical representation by default.

A k-length DNA sequence is termed a k-mer. A k-molecule (e.g. $(m, \overline{m})$) and *canonical k-mer* (e.g. $\hat{m}$) are similarly defined. Given a read set R, the set of all the k-mers of R is represented by M^k. $\hat{M}^k$ denotes the set of canonical k-mers corresponding to M^k. In order to obtain M^k or $\hat{M}^k$, each read $r \in R$ of length l is decomposed into $(l - k + 1)$ k-mers, by sliding a window of length k along the read. *Coverage* of a read set R is defined as the average number of times a base in the source DNA molecule(s) appears in R. Most widely deployed HTS instruments from Illumina are capable of generating *paired-end reads*. A paired-end read consists of two reads which are sequenced from opposite ends of a longer DNA fragment, referred to as *insert*.

3.2.2 de Bruijn Graph

While many variants of *de Bruijn graph* have been proposed for analysis of short read datasets, we use a definition that closely resembles the one in [20]. We define *de Bruijn graph* as a graph with k-molecules (or equivalently the set of canonical k-mers $\hat{M}^k$) as its vertices. An edge is included between a pair of vertices u and v, if there is a $(k-1)$ length suffix-prefix overlap between a strand of u and a strand of v. In other words, for any edge between vertices u and v, the corresponding merged $(k+1)$-mer exists in M^{k+1}. An edge incident to a vertex u of the de Bruijn graph is called an *in-edge* (resp. *out-edge*) w.r.t. u if the corresponding $(k-1)$ length overlap includes the prefix (resp. suffix) of u. Note that a vertex u can have at most four *in-edge*s and at most four *out-edge*s incident to it, and hence a maximum of eight neighbors. The key difference between the above definition of *de Bruijn graph* and the one proposed in [20] is that an undirected edge representation is sufficient for our purposes.

3.2.3 Graph Partitioning

Let $\mathcal{G} = (V, E, W_v, W_e)$ be an undirected graph. Here, V is the set of vertices, E is the set of edges, and functions $W_v : V \to \mathbb{R}$ and $W_e : E \to \mathbb{R}$ define the vertex and the edge weights respectively. In the representation of an undirected graph $\mathcal{G}$, an edge between two vertices $u \in V$ and $v \in V$ appears twice in E, as two ordered pairs (u, v) and (v, u). Let $V_1, \ldots, V_m$ be a partition of $\mathcal{G}$'s set of vertices such that $V_i \cap V_j = \emptyset$ (for $i, j \in \{1, m\}$, $i \neq j$) and $V_1 \cup \ldots \cup V_m = V$. Then, the *balance* of the partition is defined as

$$\frac{\max_{1 \leq i \leq m} C(V_i)}{\sum_{1 \leq i \leq m} C(V_i)/m}$$

where $C(V_i)$ is the sum of the weights of all vertices $v_j \in V_i$. The *cut* of the partition is defined as the total sum of weights of edges between $u \in V_i$ and $v \in V_j$ for every pair $(V_i, V_j), 1 \leq i < j \leq m$.

The *graph partitioning* problem is defined as follows: Given $\mathcal{G}$, a positive integer $m \leq |V|$, and a parameter $\epsilon > 0$, find a partition $V_1, \ldots, V_m$ of the vertices of $\mathcal{G}$ that minimizes the *cut* of the partition, while bounding the *balance* of the partition to at most $(1 + \epsilon)$. We extend the definitions of cut and balance to read dataset partitioning in Section 3.5.4.

3.3 Methodology

Table 3.1 lists key properties of the datasets used in evaluating this work. All the three datasets, namely *Fish, Bird,* and *Snake*, were provided as part of the Assemblathon 2 competition [21]. These datasets are publicly available and the instructions for obtaining them are described in [21]. The sizes of the input read sets, for all the datasets, correspond to over 50*X* coverage based on the estimated length of the respective genomes.

Table 3.1: Datasets used for experimental evaluation

Dataset	Genome length (Giga base-pairs)	Dataset size (Giga bases)	Read length (Bases)
Fish	1.0 Gbp	52.7 Gb	101
Bird	1.2 Gbp	70.7 Gb	101
Snake	1.6 Gbp	84.1 Gb	121

We ran our experiments on a cluster with a 40 Gb QDR InfiniBand interconnect. Each node in the cluster has two 2.0 GHz 8-core Intel Xeon E5-2650 processors, for a total of 16 cores per node. The 16 cores in a node share 128 GB of main memory. For our experiments, we used up to 32 nodes in the cluster, for a total of 512 cores. Further, we used distributed-memory parallel programming with OpenMPI 1.8.6. For reporting performance results, each experiment was repeated three times. The maximum wall-times from all processes were collected for each run, and the minimum times amongst the repeats of an experiment are reported as they closely represent the capabilities of the system. Other details concerning experimental methodology are provided at appropriate places in the chapter.

3.4 de Bruijn Graph Partitioning

3.4.1 Motivation

Estimating pairwise similarity among reads, for the purpose of partitioning a large set of reads, is known to be challenging both in terms of computation and memory. Therefore, we use the de Bruijn graph, constructed from the read set, to accomplish our end goal of read set partitioning. Excluding sequencing errors, the size of the de Bruijn graph is expected to be proportional to the cumulative length of the source DNA molecules from which the read set is generated. Therefore, the use of de Bruijn graph allows us to overcome both compute and memory limitations. Since reads trace paths in a de Bruijn graph, a high quality partitioning of the de Bruijn graph can be used to generate a high quality partitioning of the read set itself. Furthermore, the abundance of applications that make use of de Bruijn graph as a primary data structure, including de novo assembly, can directly leverage the partitioned de Bruijn graph.

Owing to their significance, the problems of de Bruijn graph construction and compaction received attention from several researchers. Majority of the proposed solutions are targeted at single-node machines, where multiple cores share the main memory. They are not suitable in the context of our proposed solution approach, which is meant for distributed-memory parallel systems. There are a few distributed-memory parallel solutions for compacting the de Bruijn graph. However, they cannot be leveraged due to the unique requirements of our solution approach. We need to explicitly keep track of the graph vertices, edges, and their weights. In this section, we present the data structures and the algorithms intended for constructing, compacting, and partitioning de Bruijn graphs.

3.4.2 Parallel Construction of de Bruijn Graph

We refer to a vertex u in the de Bruijn graph as a *branch* vertex if it has more than one *in-edge* or more than one *out-edge* incident to it. Otherwise, u is a *chain* vertex. If a chain

vertex u has only one neighbor or has a *branch* vertex as one of its neighbors, then u is a *terminal* vertex. In the de Bruijn graph, a *chain* or a *unitig* is a sequence of vertices $\langle c_1, \ldots, c_d \rangle$ that satisfies the following conditions: (a) c_{i+1} is a neighbor of c_i; (b) $c_i, 1 < i < d$ are chain vertices, and (c) c_1 and c_d are terminal vertices. The weight of a chain is the sum of weights of the edges between the vertices in the chain. For example, in the graph shown in Figure 3.1, v_2, v_3, v_4, v_7 and v_8 are chain vertices, whereas v_1, v_5 and v_6 are branch vertices. $\langle v_2, v_3, v_4 \rangle$ and $\langle v_7, v_8 \rangle$ are chains with weights 8 and 6 respectively.

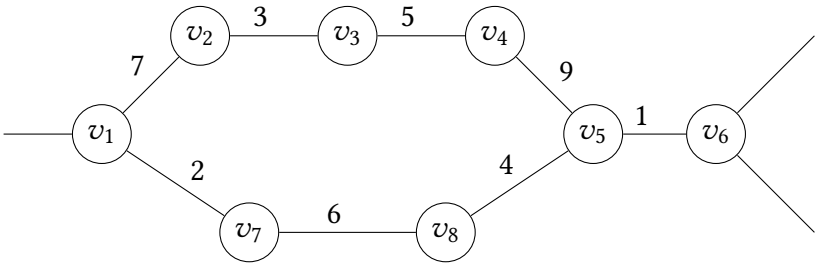

Figure 3.1: A fragment of the DBG. *in-edge*s and *out-edge*s are shown to the left and the right of a vertex respectively.

To enable efficient manipulation of the de Bruijn graph, we represent the graph as a list of tuples $((u, v), w)$, where (u, v) is an ordered pair representing the edge between vertices u and v, and w is the edge weight ***i.e.,*** the number of times the corresponding merged $(k+1)$-mer occurs in the read set. We classify an edge $((u, v), w)$ in the de Bruijn graph into one of two types: An edge (u, v) is a *chain* edge if both u and v are chain vertices. An edge (u, v) is a *branch* edge if either u or v is a branch vertex. Note that the edge type definitions are complementary to each other so that every edge in the graph belongs to exactly one of the two types. As an example, in the graph shown in Figure 3.1, $((v_2, v_3), 3)$, $((v_3, v_2), 3)$, $((v_3, v_4), 5)$, $((v_4, v_3), 5)$, $((v_7, v_8), 6)$, and $((v_8, v_7), 6)$ are chain edges, and the rest of the edges are branch edges.

Algorithm 5 constructs in parallel the de Bruijn graph for the input read set R by generating the two types of edges defined above. In Line 3, we make use of *KmerInd* [22] to build a distributed index of k-molecules. The entries in the index are of the form:

$(u, [w_1, \ldots, w_8])$. Here, the first element u is the k-molecule in its canonical representation, and the second element is an array of integers which contains, corresponding to each of u's eight possible neighbors, the number of times the respective $(k+1)$-mer occurs in R. Line 6 ensures that the frequency of occurrence of every $(k+1)$-mer in R is greater than a *threshold* parameter. This check enables filtering of low-frequency k-mers, which are produced due to sequencing errors. The value of this parameter is specified by the user.

Algorithm 5: Parallel construction of *de Bruijn graph*

Input: R, the set of reads
Output: Two distributed lists of edges: $\mathcal{C}$, *chain* edges, and $\mathcal{B}$, *branch* edges

1 **parallel** *j = processor's rank* **do**
2 Initialize $\mathcal{B}$, $\mathcal{C}$, $\mathcal{T}$ to empty list of tuples.
3 $\mathcal{I} \leftarrow$ Construct distributed index of k-molecules for R using ***KmerInd*** [22].
4 **for** $(u, [w_1, \ldots, w_8]) \in \mathcal{I}$ **do**
5 Let $v_1, \ldots, v_4$ be the k-molecules of the *in-edge* neighbors and $v_5, \ldots, v_8$ be the corresponding *out-edge* neighbors (w.r.t the alphabet set Σ). Let w_i be the weight corresponding to edge (u, v_i).
6 **for** $1 \leq i \leq 8$ **do** Set w_i to 0 if $w_i < (k+1)$-mer threshold.
7 $nbrs_i \leftarrow$ No. of non-zero elements in $[w_1, \ldots, w_4]$.
8 $nbrs_o \leftarrow$ No. of non-zero elements in $[w_5, \ldots, w_8]$.
9 **if** $((nbrs_i > 1) \vee (nbrs_o > 1))$ **then**
10 **for** $1 \leq i \leq 8$ **do** Append tuples $((u, v_i), w_i)$ and $((v_i, u), 0)$ to lists $\mathcal{B}$ and $\mathcal{T}$ respectively, if $w_i > 0$.
11 **else if** $((nbrs_i + nbrs_o) > 0)$ **then**
12 **for** $1 \leq i \leq 8$ **do** Append tuples $((u, v_i), w_i)$ to list $\mathcal{C}$, if $w_i > 0$.
13 **end**
14 **end**
15 Append $\mathcal{T}$ to $\mathcal{C}$, and sort the distributed list $\mathcal{C}$ in parallel.
16 Remove from $\mathcal{C}$ those edges that appear more than once and add them to $\mathcal{B}$.
17 **end**

Lines 7–13 in Algorithm 5 enable the segregation of edges into chain edges and branch edges, which together constitute the undirected de Bruijn graph. By counting the number of non-zero entries of the weight vector of an entry $(u, [w_1, \ldots, w_8])$ (in Lines 7 and 8), it is possible to check if u is a chain or a branch vertex. Accordingly, the tuple $((u, v_i), w_i)$ is added either to the list $\mathcal{B}$, if u is a branch vertex (in Line 10), or to the list $\mathcal{C}$, if u is a chain

vertex (in Line 12).

However, since the index is distributed, it is not possible, with only locally available information, to verify if v_i is a branch vertex or a chain vertex. This implies that after adding all the tuples (i.e, at the end of the loop at Line 13), the branch edge list $\mathcal{B}$ does not include edges $((u, v), w)$, where v is a branch vertex and u is a chain vertex, whereas the chain edge list $\mathcal{C}$ includes edges $((u, v), w)$, where v is a branch vertex. In order to update the lists $\mathcal{C}$ and $\mathcal{B}$, a temporary list $\mathcal{T}$ containing edges $((u, v), w)$ such that v is a branch vertex is created (line 10). In Lines 15– 16, by making use of $\mathcal{T}$ and parallel sorting, the edges $((u, v), w)$ in $\mathcal{C}$ with v being a branch vertex are identified and moved from $\mathcal{C}$ to $\mathcal{B}$.

Categorization into branch and chain edges allows the algorithms described in the following subsections to label the chain vertices and subsequently compact the labeled chains to generate a significantly smaller graph. Note that the distributed index of k-molecules constructed by *KmerInd* recognizes *in-edge*s and *out-edge*s separately. However, as we mentioned in Section 3.2.2, an undirected edge representation of the de Bruijn graph suffices beyond this point for our purposes. Accordingly, the output lists of Algorithm 5 are comprised of only undirected edges.

3.4.3 Parallel Compaction of de Bruijn Graph

The next step in our solution is to compact the de Bruijn graph. De Bruijn graphs constructed from read sets are known to contain several long chains. De Bruijn graph based de novo assemblers make use of this observation to identify unambiguous segments in the graph. We leverage this observation to reduce the size of the graph, thus reducing the runtime and the memory requirements for subsequent steps.

The de Bruijn graph is compacted as follows: First, using the edge lists constructed in the previous subsection, Algorithm 6 labels all the chains in the graph. Algorithm 7 then replaces vertices in every chain by their chain label. Back to the example shown in Figure 3.1, the chains $\langle v_2, v_3, v_4 \rangle$ and $\langle v_7, v_8 \rangle$ are compacted and the resulting compact

graph is shown in Figure 3.2.

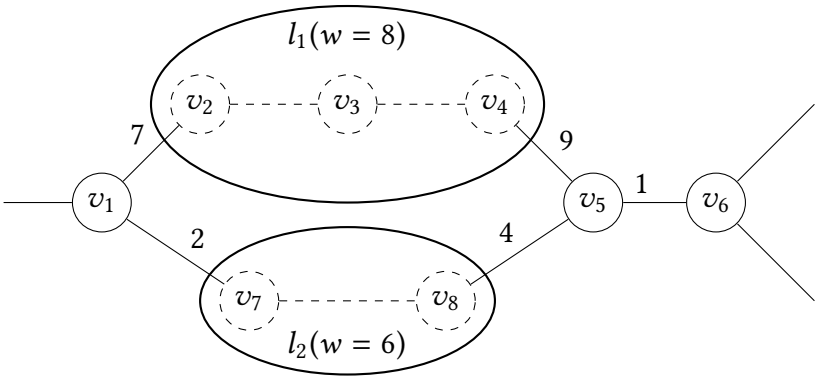

Figure 3.2: Compacted form of the de Bruijn graph fragment shown in Fig. 3.1. The chains $\langle v_2, v_3, v_4 \rangle$ and $\langle v_7, v_8 \rangle$ are replaced by labels l_1 and l_2 with weights 8 and 6 respectively.

Algorithm 6 describes the steps to label and compute weights of all the chains, where the weight of a chain is defined as the sum of the weights of all the edges that constitute the chain. Here, we make use of the fact that in the subgraph induced by only the chain edges, each chain is a connected component. This implies that labeling the chains is equivalent to labeling the connected components of the induced subgraph, which is accomplished as follows. Given the distributed list of chain edges, we use the parallel connectivity algorithm proposed in [23] (Line 3 in Algorithm 6) to identify the chain to which each chain vertex belongs to. The connectivity algorithm takes $log_2 L$ steps to complete this identification for all the chains, where L is the number of vertices in the longest chain.

After labeling all the chain vertices, the chain weights are computed as follows. In lines 4–11, a temporary distributed list of tuples $\mathcal{T}$ is created. Each tuple (u, v, l, w) in $\mathcal{T}$ corresponds to a chain edge (u, v) and includes the edge weight and u's chain label. $\mathcal{T}$ is then sorted in parallel by its label, and the total weight of the chain is computed as the sum of the weights of all tuples sharing the same label (lines 12–16). The final output from Algorithm 6 (prepared in lines 17–20) is $\mathcal{V}$, a distributed list of tuples, where each tuple corresponds to a chain vertex and contains the vertex ID (*i.e.,* the canonical k-mer representing the vertex), its chain label, and the chain weight.

Algorithm 7 uses outputs from both Algorithms 5 and 6 to complete the graph com-

Algorithm 6: Labeling of *de Bruijn graph chain* vertices

Input: Distributed list of chain edges $\mathcal{C}$ output by Algorithm 5

Output: $\mathcal{V}$, list of chain vertices. Each element in $\mathcal{V}$ is a tuple of three elements (u, l, w), where u is a chain vertex, l is the chain label, and w is the label weight.

1 **parallel** *j = processor's rank* **do**
2 Initialize $\mathcal{V}$ to an empty list of tuples.
`// Attribute chain IDs (labels) to vertices`
3 Use the connectivity algorithm proposed in [23] to assign chain IDs to vertices in $\mathcal{C}$. The output of this algorithm is $\mathcal{L}$, a distributed list of tuples (u, l). Here, u is a chain vertex and l is the ID of the chain to which u belongs to.
`// Compute chain weights`
4 Initialize $\mathcal{T}$ to an empty list of tuples. $\mathcal{T}$ accommodates tuples of four elements (u, v, l, w), where u and v are graph vertices, l is a label, and w is the edge weight.
5 **for** $((u, v), w) \in \mathcal{C}$ **do** Append tuple $(u, v, null, w)$ to $\mathcal{T}$.
6 **for** $(u, l) \in \mathcal{L}$ **do** Append tuple $(u, u, l, 0)$ to $\mathcal{T}$.
7 Parallel sort $\mathcal{T}$ by the first element of the tuple.
8 **foreach** *segment* T *of* $\mathcal{T}$ *s.t. every* $t \in T$ *has the same first element* **do**
9 $(u_t, v_t, l_t, w_t) \leftarrow$ tuple in T, whose label is not null.
10 Set l_t as the label for all the tuples of T
11 **end**
12 Parallel sort $\mathcal{T}$ by the label element of the tuple.
13 **foreach** *segment* T *of* $\mathcal{T}$ *s.t. every* $t \in T$ *has the same label* **do**
14 $w_s \leftarrow$ sum of the weight values of all the tuples in T.
15 Set w_s as the weight for all the tuples of T.
16 **end**
`// Generate output list`
17 Parallel sort $\mathcal{T}$ by the first element of the tuple.
18 **foreach** *segment* T *of* $\mathcal{T}$ *s.t. every* $t \in T$ *has the same first element* **do**
19 Let (u_t, v_t, l_t, w_t) be the first element of T.
20 Append (u_t, l_t, w_t) to $\mathcal{V}$.
21 **end**
22 **end**

paction. It outputs, for each edge in the compact de Bruijn graph, a tuple (u, v, we, wu), where u and v are the vertices in the compact graph, we is the edge weight and wu is the vertex weight of u. To construct the compact de Bruijn graph, Algorithm 7 starts with the distributed list of branch edges $\mathcal{B}$ from Algorithm 5. For each edge $((u, v), w) \in \mathcal{B}$, a tuple in the output format with zero vertex weight i.e., $(u, v, w, 0)$, is added to $\mathcal{E}$ (Line 3 in Algorithm 7). Recall that in a branch edge (u, v), either u or v should be a branch vertex, which in turn implies that either v or u can potentially be a chain vertex. Such entries in $\mathcal{E}$ are identified and the chain vertex is replaced by its corresponding label.

Lines 4–11 replace the vertex u in the tuples by its label, if u is a chain vertex. The vertex weights are updated with the corresponding label (chain) weights. Similarly, Lines 12—18 replace the vertex v by its label, if v happens to be a chain vertex. In both cases, the distributed list of vertex-label mapping $\mathcal{V}$ constructed in Algorithm 6 is used to match the vertex IDs. Finally, in Lines 19–23, the weights for edges and vertices of the compact graph are assigned.

3.4.4 Partitioning of Compacted de Bruijn Graph

We treat the compact de Bruijn graph output by Algorithm 7 as a standard graph data structure (as defined in Section 3.2.3) and find a partition for this graph. Graph partitioning is an important problem with applications in several domains and hence, has received significant attention from researchers over the years [24]. While the details of the various solution approaches proposed differ from one another, most of them adopt the multilevel graph partitioning paradigm first introduced in [25]. The key idea is to recursively coarsen the graph to reduce its size, partition the coarsest graph, and recursively uncoarsen the graph to its original size. Refinement is typically performed after every uncoarsening to improve the *cut*, defined in Section 3.2.3. Graph partitioning algorithms differ in the exact techniques they adopt for coarsening, partitioning, and uncoarsening.

We use a recently proposed distributed-memory graph partitioning algorithm called

Algorithm 7: Parallel compaction of *de Bruijn graph*

Input: (a) $\mathcal{B}$, list of branch edges, output from Algorithm 5, and (b) $\mathcal{V}$, list of labeled chain vertices, output from Algorithm 6. Both lists are distributed.

Output: Compact *de Bruijn graph* as a distributed list of edges, $\mathcal{E}$. $\mathcal{E}$ includes tuples with four elements (u, v, we, wu), where u and v are the vertices of the compact graph, we is the edge weight, and wu is the node weight of u.

parallel *j = processor's rank* **do**
  Initialize $\mathcal{E}$ to an empty list of tuples.
  for $((u, v), w) \in \mathcal{B}$ **do** Append $(u, v, w, 0)$ to $\mathcal{E}$.
  `// Replace chains in the graph by their labels`
  for $(u, l, w) \in \mathcal{V}$ **do** Append $(u, l, null, w)$ to $\mathcal{E}$.
  Parallel sort $\mathcal{E}$ by the tuple's first element.
  foreach *segment* E *of* $\mathcal{E}$ *that share the same first element* **do**
    $(u_t, l_t, null, wu_t) \leftarrow$ tuple t in E whose third element is null.
    Set the first element of all the tuples in E to l_t.
    Set the fourth element of all the tuples in E to wu_t.
    Remove t from $\mathcal{E}$.
  end
  for $(u, l, w) \in \mathcal{V}$ **do** Append $(l, u, null, w)$ to $\mathcal{E}$.
  Parallel sort $\mathcal{E}$ by the tuple's second element.
  foreach *segment* E *of* $\mathcal{E}$ *that share the same second element* **do**
    $(l_t, u_t, null, wv_t) \leftarrow$ tuple t in E whose third element is null.
    Set the second element of all the tuples in E to l_t.
    Remove t from $\mathcal{E}$.
  end
  `// Update weights`
  Parallel sort $\mathcal{E}$ by the first element.
  foreach *segment* E *of* $\mathcal{E}$ *that share the same first element* **do**
    $sw \leftarrow$ sum of all the edge weights of $t \in E$.
    for $t \in E$ **do** Add sw to wu of t.
  end
end

KaHIP [26] to partition the compacted de Bruijn graph output by Algorithm 7. *KaHIP* makes use of size-constrained label propagation algorithm during coarsening and uncoarsening phases, and coarse-grained evolutionary algorithm for partitioning the coarsest graph. In the context of the multilevel graph partitioning framework, the labeling of de Bruijn graph chains proposed in Algorithm 6 enables rapid compaction and un-compaction of the de Bruijn graph. As we will demonstrate in Section 3.4.5, this allows *KaHIP* to generate partitions significantly faster than if they were generated using the original de Bruijn graph. Use of domain knowledge about sequencing read sets and de Bruijn graphs enables realization of this efficient approach.

3.4.5 Results and Analysis

In this section, for the three datasets listed in Table 3.1, we present quality and runtime results for the parallel algorithms to construct, compact, and partition a de Bruijn graph.

First, we evaluate the impact of compaction on reducing the size of the de Bruijn graph. Table 3.2 shows the compaction ratio obtained for all three datasets. It is computed as a ratio of the number of vertices in the de Bruijn graph before compaction to the number of vertices in the compact de Bruijn graph. We obtain a mean compaction ratio of 47 across the three datasets, implying that our proposed approach significantly reduces the size of the de Bruijn graph. For constructing the initial de Bruijn graph, we used a value of 5 for the *threshold* parameter. This implies that a k-molecule u is filtered out, if all edges incident to u have a weight ≤ 4. For the parameter k, we used a value of 31. We used *KmerGenie* software [27] to estimate the optimal k-mer length. However, both of these parameters can be specified by users. Note that the compact de Bruijn graph produced by our algorithm can be used by de novo assemblers for rapid *unitig* (unambiguous segment) generation in distributed-memory parallel settings.

Next, we present the *cut* and the *balance* values for the de Bruijn graph partitions generated by our algorithm. These measures, defined earlier in Section 3.2.3, serve as the quality

Table 3.2: Reduction in the size of the de Bruijn graph due to compaction

Dataset	Plain graph (No. of vertices)	Compacted graph (No. of vertices)	Compaction ratio
Fish	733,774,187	16,672,988	44
Bird	1,208,521,390	25,740,770	47
Snake	1,361,026,568	27,199,895	50

indicators of a graph partition. A good graph partition has low cut value and a balance value close to 1. Table 3.3 shows the *cut* ratios computed for the partitions corresponding to the de Bruijn graphs of the three datasets. The *cut* ratio refers to the ratio of the sum of weights of edges cut to the sum of weights of all edges in the graph. The low cut ratio obtained for all the three datasets demonstrates that our solution produces high quality partitioning of de Bruijn graphs. All the partitions have a *balance* value of 1.03.

Table 3.3: Quality of de Bruijn graph partitioning

Dataset	Sum of weights of all edges in the graph	Sum of weights of edges cut	*Cut* ratio
Fish	13,593,910,042	19,252,245	1.42×10^{-3}
Bird	22,462,771,436	22,337,839	0.99×10^{-3}
Snake	29,754,489,857	47,197,297	1.59×10^{-3}

One of the input parameters for the graph partitioning algorithm is the number of desired partitions. One way to choose this parameter is to set it to the number of available compute nodes in the distributed system. However, if there are thousands of distributed nodes available in a system, then the output graph partitions will be too fragmented, which in turn increases the communication cost required to transfer neighboring regions of the graph among the nodes. Therefore, this parameter needs to be carefully selected for a given dataset.

We choose the number of partitions based on the available memory per node, as follows. Suppose, L_G is the approximate total length of the source DNA molecule(s), in Gbp, corresponding to a read dataset, and mem_b bytes are required for analysis, per base of the

genome. Then, to analyze the dataset, $(L_G \times mem_b)$ GB of memory is required. However, if mem_n is the size of memory available per node in the distributed system, expressed in GB, at least $n_p = \left\lceil \frac{L_G \times mem_b}{mem_n} \right\rceil$ number of nodes are required to run the analysis. We select the number of desired partitions to be n_p because n_p partitions are sufficient to run the analysis using n_p nodes in the distributed system. This does not prevent the user from running the analysis using more than n_p compute nodes in the system. To run the analysis with more than n_p nodes, a partition is duplicated on multiple nodes and only a fraction of the partition is analyzed in a node. For example, to run the analysis with 128 partitions using 1024 nodes, each partition is duplicated in 8 nodes and each node processes only $1/8$-th of the locally available partition. Such a duplication scheme incurs far less communication cost compared to processing as many partitions as the number of available nodes. With a conservative assumption of $mem_n = 32$ and $mem_b = 64$, the number of partitions for Fish, Bird, and Snake datasets are computed as 106, 142, and 168, respectively.

Finally, we present runtime and parallel scalability results of Algorithms 5, 6, and 7. Table 3.4 lists the runtime in seconds incurred by each of the algorithms individually (columns 2-4) and also the total runtime (column 5), for the Bird dataset. We present results by varying the number of available cores from 64 to 512. Note that the number of MPI processes used is the same as the number of cores. Our algorithms demonstrate good scalability across the board. The discrepancy in runtime (most notable in case of 64 cores) is due to interference from concurrently running jobs, while accessing the interconnection network and the parallel file system. With 512 cores, our solution is able to compute the compacted de Bruijn graph for a read set of size 71 Gbp in 2.7 minutes. We refer the reader to [26] for a demonstration of the parallel scalability of *KaHIP*. In our experiments, *KaHIP* took 4.3 minutes to partition the compacted de Bruijn graph for the Bird dataset using 512 cores. We also attempted to partition the initial (*i.e.,* un-compacted) de Bruijn graph generated by Algorithm 5 directly. *KaHIP* spent an order of magnitude more time to partition the initial de Bruijn graph, validating the merit behind our approach of compacting the de Bruijn

graph prior to partitioning the graph.

Table 3.4: Runtime in seconds for the *Bird* dataset for de Bruijn graph construction (Algorithm 5), chain labeling (Algorithm 6) and compaction (Algorithm 7).

No. of cores	Algorithm 5 (s)	Algorithm 6 (s)	Algorithm 7 (s)	Total (s)
64	391	790	33	1214
128	159	309	11	479
256	76	180	6	262
512	45	115	3	163

3.5 Read Dataset Partitioning

3.5.1 Motivation

Algorithms 5, 6, and 7 described in the previous section facilitate partitioning of the de Bruijn graph corresponding to a read set R. Further, we demonstrated that the partitions generated are of very high quality. Since reads trace paths in the de Bruijn graph, high quality partitioning of the de Bruijn graph can be translated into a high quality partitioning of R. In this section, we describe the algorithm we developed for this purpose. The intuition behind our algorithm is to let the k-mers in a read vote for the partition to which the read should be assigned. We also describe a parallel algorithm we developed for evaluating the quality of read set partitioning and present corresponding results.

3.5.2 Parallel Partitioning of Reads

The parallel algorithm we designed for generating a partition for the read set based on the de Bruijn graph partitioning takes as its input the following: (a) the read set R, (b) the partition IDs assigned by the graph partitioner to vertices of the compact de Bruijn graph, (c) the list of de Bruijn graph vertices, and (d) the vertex to chain label mapping determined by Algorithm 6. The algorithm starts off by propagating the partition IDs assigned to the label vertices of the compact graph to the canonical k-mers that make up the corresponding

chains. Using the mapping from canonical k-mers to partition IDs, a distributed index $\mathcal{D}$ is then constructed. Finally, the distributed index is used to compute the partition ID for a read r as the most frequently assigned partition ID for the k-mers in read r. In case of a *paired-end read*, our approach assigns the same partition ID to both reads. We accomplish this by subjecting the first read to the read partitioning algorithm and assigning the resulting partition ID to the second read as well.

Algorithm 8 describes our approach for generating a partition for the read set based on the de Bruijn graph partitioning. In Algorithm 8, Lines 3–6 map the partition IDs to the vertex IDs of the compact de Bruijn graph. This is accomplished by constructing $\mathcal{K}$, a distributed list of tuples (v, pid), where v is a vertex of the compact de Bruijn graph and pid is the partition ID computed by the partitioner. Recall that in Section 3.4.3, we constructed the compact graph by replacing the chain vertices by the corresponding chain labels. In Lines 7–14, partition IDs assigned to the label vertices of the compact graph are propagated back to the canonical k-mers that make up the corresponding chains.

In Lines 15–25, we compute the partition IDs for all the reads, taking a batch of B reads at a time. Batching helps overcome any memory size constraints and based on the available memory capacity, the value of B can be specified by the user as an input parameter. Using the mapping from canonical k-mer to partition ID, available as $\mathcal{T}$ at the end of Line 14, a distributed index $\mathcal{D}$ is constructed in Line 15. A local lookup table of partition identifiers for the current batch of canonical k-mers is constructed in Line 19 from the results of the query to $\mathcal{D}$ (Line 18). This local lookup table is necessary because copies of elements in a query submitted to the distributed index are collapsed prior to the result being returned. The lookup table is then used in Lines 20–24 to compute the partition ID for a read r as the most frequently assigned partition ID for the k-mers in read r.

Algorithm 8: Parallel generation of partitioning of a read set

Input: (a) Read set R, (b) Partition of the compacted DBG, (c) Vertex list, and (d) Vertex-chain (label) mapping.

Output: Partition identifier for every read in R.

parallel *j = processor's rank* **do**
  Initialize $\mathcal{K}$, $\mathcal{T}$ to empty list of tuples.
  $\mathcal{P} \leftarrow$ partition records of the compact de Bruijn graph corresponding to processor j.
  $\mathcal{V} \leftarrow$ Vertex list corresponding to processor j.
  Re-distribute both $\mathcal{P}$ and $\mathcal{V}$ s.t. each processor has approx. the same number of records. Note that after re-distribution $|\mathcal{P}| = |\mathcal{V}|$ in every processor.
  for $i \leftarrow 1$ **to** $|\mathcal{P}|$ **do** Append $(\mathcal{V}[i], \mathcal{P}[i])$ to $\mathcal{K}$.
  $\mathcal{L} \leftarrow$ Vertex-chain mapping pairs for proc. j.
  for $(u, pid) \in \mathcal{K}$ **do** Append (u, u, pid) to $\mathcal{T}$.
  for $(u, l) \in \mathcal{L}$ **do** Append $(l, u, null)$ to $\mathcal{T}$.
  Parallel sort $\mathcal{T}$ by the tuple's first element.
  foreach *segment T of $\mathcal{T}$ having the same first element* **do**
    $(l, u, pid) \leftarrow$ tuple in T with non-null third element.
    Set the third element to pid for all tuples in T.
  end
  $\mathcal{D} \leftarrow$ Construct a distributed index of (u, pid) from $(l, u, pid) \in \mathcal{T}$.
  for *each batch B of the read set R* **do**
    $K \leftarrow$ Set of canonical k-mers w.r.t. the reads in B.
    Query in $\mathcal{D}$ the partition identifiers for all $k \in K$.
    $D \leftarrow$ local lookup table of $pids$ for all $k \in K$.
    for $r \in B$ **do**
      $M \leftarrow pids$ queried from D for all k-mers in r.
      $rpid \leftarrow$ most frequently assigned pid in M.
      r is assigned the partition identifier $rpid$.
    end
  end
end

3.5.3 Partitioning Quality Evaluation

If the true sequence of the source DNA molecule and the starting position for each read in the source DNA sequence are known, then it is possible to evaluate the quality of read partitioning against the truth. A good partitioning of the reads would place the reads derived from the same region in the source DNA within the same partition. However, even if the true source sequence is available, it is not possible to identify, for every read r, the exact position of r in the source sequence. Therefore, for the read sets given in Table 3.1, we use the local alignment of the reads against the reference genome as an approximation of the truth and evaluate the quality of the read partitioning against it.

We developed a parallel algorithm to evaluate the quality of a read partition using the alignment information of the read set R. It takes as its input the partition IDs assigned to reads by the proposed method and a SAM file, the standard file format to specify alignment data for a read set. We measure the partition quality using two measures called `intra-pairs` and `inter-pairs`, corresponding to the number of overlapping read pairs (as inferred from their alignment to the reference genome) that are assigned the same and different partition IDs, respectively. We consider two reads to overlap if they share at least one base in the reference. Note that this is a rather stringent requirement as no algorithm using only read information can reliably detect short overlaps, let alone a single base overlap. For every overlapping read pair, we determine if both constituent reads are placed in the same partition (`intra-pairs`) or in different partitions (`inter-pairs`).

In Algorithm 9, an alignment record (implemented as a C++ `struct`) has the following fields: Read identifier (`rid`), Reference name (`ref`), Reference position (`pos`), Alignment score (`score`), Alignment flag (`flag`) and Partition identifier (`pid`). The parallel algorithm reads the SAM records and partition IDs from separate files, in a distributed manner. Some of the reads in R can map to multiple positions in the reference sequence(s). Such reads have more than one record in the SAM file. Lines 5 and 6 in the algorithm aid in identifying the records corresponding to such reads. For every such read, we replace

the corresponding records with one record indicating an ambiguous mapping. Now, we are left with exactly one SAM record and one partition ID for each read in the lists $\mathcal{A}$ and $\mathcal{P}$, respectively. Redistribution of both lists allows the corresponding records to be paired with each other (Line 10). Since it is not possible to reliably know where reads with ambiguous mapping belong to, we exclude them from evaluation process along with unmapped reads (Line 11). Lines 12–28 help us identify pairs of reads that overlap with each other, as deciphered from their alignment to the reference. We consider two reads to overlap if they share at least one base in the reference. For every such pair, we determine if both are placed in the same partition (`intra-pairs`) or different partitions (`inter-pairs`). Finally, we compute cumulative statistics across all processors and report them (Line 29).

3.5.4 Results and Analysis

We present results pertaining to the partitioning of the read sets. In order to evaluate the quality of read set partitioning, we make use of the local alignment of the reads to the reference sequence. For the three datasets used in our experiments, we used the submissions that were ranked the best in the Assemblathon 2 competition as the reference sequence [21]. We then mapped the read datasets to the corresponding reference sequence using BWA local alignment software [28]. We allowed an error rate of 2% while mapping the reads.

In order to enable interpretation of read partitioning quality results, we extend the definitions of *cut* and *balance*, defined for graph partitioning in Section 3.2.3, to read partitioning. We define *balance* of a read partitioning as the ratio of the size of the largest read partition to the average size of the partitions. *Cut* is defined as the number of overlapping pairs of reads that are assigned to different partitions (`inter-pairs`). *Cut ratio* is defined as the ratio of *cut* to the total number of overlapping read pairs *i.e.,* the sum of `intra-pairs` and `inter-pairs`. A lower cut ratio implies that fewer overlapping read pairs are assigned to different partitions, indicating a better quality partition.

Table 3.5 shows the total number of overlapping read pairs *i.e.,* `intra-pairs`+`inter-pairs`

Algorithm 9: Algorithm for evaluating read partitioning quality

Input: (a) Partition IDs for reads in set R, (b) Alignment records for reads in set R provided in SAM format, and (c) L, Read length.

Output: Evaluation statistics: `intra-pairs` and `inter-pairs`.

parallel $j =$ *procesor's rank* **do**

`// Load Alignment Records`

$F_S \leftarrow$ Size of the input SAM file.

$lf \leftarrow \left\lfloor \frac{F_S j}{p} \right\rfloor + 1; rf \leftarrow \left\lfloor \frac{F_S (j+1)}{p} \right\rfloor$.

$\mathcal{A} \leftarrow$ Alignment records starting in the region $[lf, rf]$ of the input SAM file; For each alignment, `rid`, `ref`, `pos`, `score` and `flag` fields are populated.

Parallel sort $\mathcal{A}$ by `rid`.

Replace $\mathcal{A}$ records (count> 1) corresponding to the same `rid` with a single record whose `flag` indicates ambiguity. Redistribute $\mathcal{A}$.

`// Load Partition Data`

$F_P \leftarrow$ Size of the file with partition IDs for reads in R.

$lp \leftarrow \left\lfloor \frac{F_P j}{p} \right\rfloor + 1; rp \leftarrow \left\lfloor \frac{F_P (j+1)}{p} \right\rfloor$.

$\mathcal{P} \leftarrow$ Partition records starting in the region $[lp, rp]$. Redistribute $\mathcal{P}$.

Copy `pid` from $\mathcal{P}$ to the corresponding $\mathcal{A}$ record's `pid` field.

`// Filter Bad Alignments`

Eliminate ambiguous and unmapped alignments in $\mathcal{A}$ as indicated by the `flag` field.

`// Evaluation Loop`

Sort $\mathcal{A}$ by `ref`, followed by `pos`.

$L_j \leftarrow |\mathcal{A}|$.

$A_r \leftarrow$ Snd/Rcv the last record in $\mathcal{A}$ to proc. $((j+1)p)$.

$T_r \leftarrow$ Rcv/Snd from proc. $((j+1)p)$ those records in its $\mathcal{A}$ whose `ref` $= A_r$.`ref` and `pos` $\leq (A_r$.`pos`$) + L - 1$.

Append T_r to $\mathcal{A}$.

`intra-pairs` $\leftarrow 0$; `inter-pairs` $\leftarrow 0$.

for $i \leftarrow 1$ **to** L_j **do**

  $k \leftarrow (i+1)$.

  while $k \leq |\mathcal{A}|$ *and* $\mathcal{A}[i]$.`ref` $= \mathcal{A}[k]$.`ref` *and* $\mathcal{A}[k]$.`pos` $\leq \mathcal{A}[i]$.`pos` $+ L - 1$ **do**

    if $\mathcal{A}[k]$.`pid` $= \mathcal{A}[i]$.`pid` **then**

      `intra-pairs` $\leftarrow$ `intra-pairs` $+ 1$.

    else

      `inter-pairs` $\leftarrow$ `inter-pairs` $+ 1$.

    end

    $k \leftarrow (k+1)$.

  end

end

Add all `intra-pairs` and `inter-pairs` using a parallel reduction to processor 0 and report them.

end

(column 2), `inter-pairs` (column 3), and cut ratio (column 4) for all the three datasets. The cut ratio shown in the table corresponds to the reads that are error-free *i.e.,* they map to the reference sequence with no errors. Cut ratio corresponding to 2% error rate (*i.e.,* all reads that map with up to 2% error rate are considered) for Fish, Bird, and Snake datasets is 2.15×10^{-2}, 1.42×10^{-2}, and 5.11×10^{-2}, respectively. The results demonstrate that a high quality partitioning of the de Bruijn graph translates into a high quality partitioning of the corresponding read set. Balance for the read partitioning of Fish, Bird, and Snake datasets is 1.02, 1.03, and 1.10, respectively.

Table 3.5: Read partitioning quality evaluation for all datasets

Dataset	No. overlapping read pairs	`inter-pairs`	Cut ratio
Fish	20,930,646,131	193,013,621	0.92×10^{-2}
Bird	14,302,047,674	304,849,664	2.13×10^{-2}
Snake	6,728,041,314	263,882,542	3.92×10^{-2}

Finally, we present runtime and parallel scalability results for the algorithm used to compute read partitioning from de Bruijn graph partitioning. The results shown in Table 3.6 for Bird dataset demonstrate that the algorithm scales almost perfectly and takes less than 3.8 minutes using 512 cores. The end-to-end runtime for the Bird dataset using 512 cores is around 11 minutes. Therefore, our solution not only produces high quality partitions but is also fast and scalable.

Table 3.6: Runtime in seconds for the *Bird* dataset for computing read partitioning from de Bruijn graph partitioning.

No. of cores	64	128	256	512
Runtime (s)	2090	950	456	226

3.6 Discussion

An alternative approach for partitioning a set of reads is to explicitly construct a read-similarity graph, where each vertex is a read and an edge between a pair of vertices implies similarity between the corresponding pair of reads. To evaluate the similarity between a pair of reads, a wide variety of measures have been proposed ranging from pair-wise alignments to alignment-free techniques [29], trading off accuracy for computational complexity. While the approach of read-similarity graph appears to be intuitive, constructing this graph using even the simplest of the available measures is computationally challenging. Partitioning the resulting graph is even more challenging owing to its size. In contrast, our proposed solution does not suffer from these challenges. This is because the size of the de Bruijn graph is expected to be a function of the length of the reference sequence rather than the size of the read set. Chains in the de Bruijn graph are amenable for easy identification and rapid compaction. Furthermore, the size of the compacted graph enables fast graph partitioning. The algorithm we proposed for evaluating the quality of read dataset partitioning can be used to evaluate the efficacy of alternative approaches for generating read set partitioning.

There are two prior works that discuss partitioning of read sets [30, 31], but with very different objectives that are less applicable in our context. In these works, the goal is not about partitioning the read sets into uniformly sized partitions. In [30], clusters of reads that give rise to distinct connected components in the de Bruijn graph are referred to as partitions. No restriction is imposed on the size of any single cluster. This poses a significant limitation for parallel processing, as the size of the largest cluster dictates the lower bound on the runtime. In [31], reads that potentially belong to a region of interest in the reference sequence are identified from a HTS dataset and segregated for subsequent targeted analysis, which the authors refer to as partitioning. Therefore, our work stands as the first demonstration of partitioning large-scale HTS read datasets to facilitate parallel

genomics analyses.

After developing the proposed framework, we made use of openly available, efficient solutions for solving some subproblems in the framework, namely constructing a distributed index of k-molecules, computing the connectivity, and partitioning general graphs. Researchers have spent significant effort over the years to develop and improve solutions to these problems. By making use of the state-of-the-art methods available to solve these problems as components of our approach, we are able to devise an effective overall solution. In addition, the individual components can be replaced with better methods as they become available.

CHAPTER 4

PARALLEL REFERENCE-BASED COMPRESSION OF HTS READ DATASETS

4.1 Introduction

Since the introduction of high-throughput sequencing (HTS) machines, the cost of sequencing has been declining and the throughput has been increasing exponentially [2]. For instance, using the recent NovaSeq line of instruments from Illumina, the current market leader, sequencing cost is expected to come down to $100 per human genome. Transmission, storage, and archival of HTS short read datasets pose significant challenges owing to the large size of such datasets. Constant improvements to sequencing technology, in the form of increasing throughput and decreasing cost, and its growing adoption for a wide variety of applications amplify the problem. In response to this problem, researchers resorted to compression of read datasets.

General-purpose compression algorithms have been widely adopted for representing HTS read datasets in a compact form. Read datatsets have several unique properties that make it difficult for general-purpose compressors to fully exploit the redundancy present in these datasets. Domain-specific properties of read datasets include reduced size of alphabet, interleaved streams of data, fixed length for reads, occurrence of reads and their reverse complements, paired representation of reads, scattered nature of redundancy, and availability of reference sequences. Researchers proposed special-purpose compression algorithms, that exploit one or more of these properties, to improve upon the compression efficiency of general-purpose compressors. Based on whether or not a reference sequence is made use of during compression, specialized compressors can be classified as reference-based or reference-free, respectively.

In this work, we leverage all of the above mentioned domain-specific properties of read

datasets to develop ParRefCom, a parallel reference-based compressor for genomics read datasets. Read datasets generated using HTS instruments widely deployed today typically contain what are known as *paired-end* reads. A paired-end (PE) read is comprised of two separate but related reads, and the pairing information can serve to be crucial during biological analysis. Our specialized compressor allows the following lossless transformations of PE reads in the datasets : reordering of reads while keeping paired ends together and reordering of individual reads within a PE read. By exploiting these insights, ParRefCom is able to significantly improve upon the compression efficiency over state-of-the-art. More specifically, for a benchmark human dataset, the size of the compressed output is 21% smaller than that produced by SPRING [32], the current best algorithm.

At a high-level, our solution approach consists of the following steps: (1) Specialized-alignment of PE reads to standard reference, (2) Categorizing PE reads based on the number of ends aligned, and (3) Customizing compression strategies for reads in different categories. In this work, we develop fast and scalable parallel algorithms for accomplishing each of these tasks. We demonstrate that ParRefCom achieves superior compression efficiency compared to existing methods. Our compressor is asymmetric by design - decompression speed is about an order of magnitude faster than compression speed. This asymmetricity in design goes well with the real world requirement of compressing a dataset once and decompressing (using) it many times.

Reference-based compression algorithms tend to achieve superior compression efficiency as they are able to leverage external knowledge in the form of reference sequence. On the other hand, reference-free compression algorithms tend to achieve superior compression speed as they avoid the computationally expensive step of base-to-base alignment. As ParRefCom makes use of a fast and scalable specialized alignment algorithm, it combines the best of both worlds and offers high compression efficiency and speed as depicted in Figure 4.1.

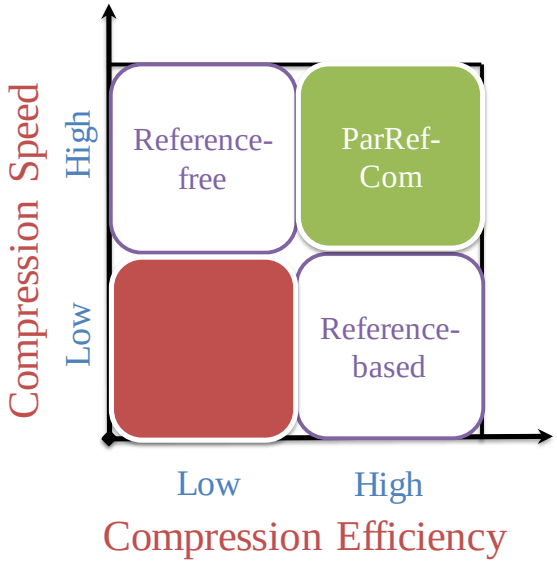

Figure 4.1: ParRefCom provides the efficiency benefits of reference-based compression as well as the speed benefits of reference-free compression

4.2 Background

4.2.1 Sequencing and Representation

A DNA molecule is comprised of two strands. The forward strand of the molecule, defined in the $5'$ to $3'$ direction, is modeled as a sequence of characters from the alphabet set $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Given such a sequence of l characters, $s = s_1, \ldots, s_l$, its reverse complementary strand *i.e.,* the sequence in the $3'$ to $5'$ direction, is $\overline{s} = c(s_l), \ldots, c(s_1)$, where $c(x), x \in \Sigma$ is the mapping function : $c(\mathtt{A}) \rightarrow \mathtt{T}, c(\mathtt{C}) \rightarrow \mathtt{G}, c(\mathtt{G}) \rightarrow \mathtt{C}$, and $c(\mathtt{T}) \rightarrow \mathtt{A}$. High-throughput sequencing (HTS) instruments are used to sequence a large number of randomly generated fragments from the genome. A few hundred bases are sequenced from these fragments and are commonly referred to as *reads*.

Most widely deployed HTS instruments from Illumina are capable of generating *paired-end reads*. A paired-end read consists of two reads which are sequenced from opposite ends of a DNA fragment, referred to as *insert*. Further, it is typical to sequence one of the reads from the forward strand and the other from the reverse complementary strand.

Pairing information can serve to be crucial during biological analysis. HTS instruments, owing to their limitations, fail to accurately decipher bases sometimes. When inference is ambiguous or unsuccessful, an `N` character is recorded in the read. For this reason, the DNA alphabet set is expanded to include `N` for read datasets and $c(\texttt{N}) \rightarrow \texttt{N}$. Each base in the genome is typically spanned by multiple reads. This oversampling or redundancy is required to facilitate subsequent analysis of the reads. *Coverage* is defined as the average number of reads spanning a base in the genome. A k-length DNA sequence is termed a k-mer.

4.2.2 FASTQ File Format

Read datasets generated using HTS instruments are typically represented as FASTQ files [33]. A FASTQ file contains the following pieces of information for every single-end read : Read identifier, Bases constituting the read, Comment line, and Quality score corresponding to each base. Read identifiers are typically not made use of during analysis. Further, due to their structure, it is relatively straightforward to represent the identifiers compactly. Prior works explored such representations. Comments are either empty or exactly identical to identifier lines. Significant efforts were devoted to develop standalone compressors for quality scores. A recent work explored use of a single bit to capture a quality score value [34]. Further, the work demonstrated that such lossy compression of metadata does not have any noticeable effect on biological analysis. Due to these reasons, in this work, we focus on compactly representing the read data in FASTQ files in a lossless manner.

4.2.3 General-purpose Vs. Special-purpose Compression

General-purpose compression algorithms have been widely adopted for representing HTS read datasets in a compact form, with ZIP family of algorithms and their blocked variants being the prominent examples [35]. Domain-specific properties of read datasets make it difficult for general-purpose compressors to fully exploit the redundancy present in these

datasets. Alphabet size of characters occurring in reads is small, typically 5. FASTQ files contain several interleaved streams of data as described in Section 4.2.2. Reads generated by HTS instruments mostly have fixed length. Reads and/or their reverse complements can represent segments of DNA in datasets. Reads are represented as a related pair in case of paired-end read datasets. Redundancy in sequencing, necessary to facilitate subsequent analysis of reads, is scattered across the dataset. Differences between genomes of organisms belonging to a species are typically very small. Therefore, genome of an organism of a species can be made use of to compactly represent the read dataset of another organism belonging to the same species. Researchers proposed special-purpose compression algorithms, that exploit one or more of the above properties, to improve upon the compression efficiency of general-purpose compressors. Based on whether or not a reference sequence is made use of during compression, specialized compressors can be classified as reference-based or reference-free, respectively. In the following section, we furnish a survey of special-purpose compression algorithms proposed for HTS short read datasets.

4.3 Related Work

A number of special-purpose compression tools have been proposed over the years for compression of FASTQ datasets, both in reference-free and reference-based categories. These include DSRC [36], Fqzcomp [37], Fastqz [37], FQC [38], SCALCE [39], LW-FQZip [40], Quip [41], Leon [42], k-Path [43], and Mince [44]. A recent review article evaluated tools with publicly available implementations using a set of benchmark datasets [45]. The review article also provides short descriptions of the tools mentioned previously. Three special-purpose compression tools, which demonstrated better compression efficiency, have come out since the review article was published - FASTORE [34], minicom [46], and SPRING [32].

FASTORE clusters reads, i.e. distributes them into bins, such that reads from neighboring positions are likely to belong to the same cluster. Within each bin, reads are matched

to generate a *reads similarity graph*. After this, reads go through an optional step of re-distribution and matching. Reads are then assembled into contigs using the final similarity graph. Based on the outcome of the matching, reads are encoded with respect to contigs or other reads. minicom also takes an approach similar to that of FASTORE. It additionally attempts to merge contigs to build longer contigs. The current best specialized compressor, in terms of compression efficiency, is SPRING. SPRING tool comprises of the following steps : Reordering reads according to their position in the genome using *hashed substring indices*, Encoding reordered reads to remove redundancy between consecutive reads, and Compressing encoded reads using general purpose compression tools. In the following section, we provide a high-level overview of our solution approach.

4.4 Overview of Solution Approach

In this work, we leverage all of the previously mentioned domain-specific properties of read datasets to develop a reference-based compressor for genomics read datasets. The ordering of paired-end (PE) reads in a dataset and the ordering of two reads of a PE read do not carry any special significance. Therefore, our specialized compressor allows the following lossless transformations of PE reads in datasets : reordering of reads while keeping paired-ends together and reordering of individual reads within a PE read. By exploiting these insights, ParRefCom is able to significantly improve upon the compression efficiency over state-of-the-art.

We developed a solution approach that leverages all of the properties described in Section 4.2.3 and the insights mentioned above. A high-level description of the overall approach is as follows. First, we perform a specialized alignment of PE reads in the dataset using standard reference for the species. Next, we classify the PE reads based on the outcome of the alignment. The categories are : Two-aligned PE read (when both ends of the PE read align), One-aligned PE read (when one of the ends of the PE read aligns but the other does not), and Non-aligned read (when both ends of the PE read do not align).

Finally, we develop custom compression strategies for each of these categories. Through evaluation using an assortment of read datasets, we demonstrate that PE reads in the first category (Two-aligned) significantly outnumber those in the remaining two categories.

In the output of the specialized compressor, we capture One-aligned and Non-aligned PE reads as they are. For Two-aligned PE reads, we capture the following pieces of information in place of the reads themselves : (1) Starting location with respect to the reference sequence, (2) Number of differences with respect to the reference sequence, (3) Positions of differences within a read, (4) Bases corresponding to differences within a read and (5) Locations of other ends for one of the ends. Collectively, these pieces of information are sufficient to reconstruct the original PE reads as depicted in Figure 4.2.

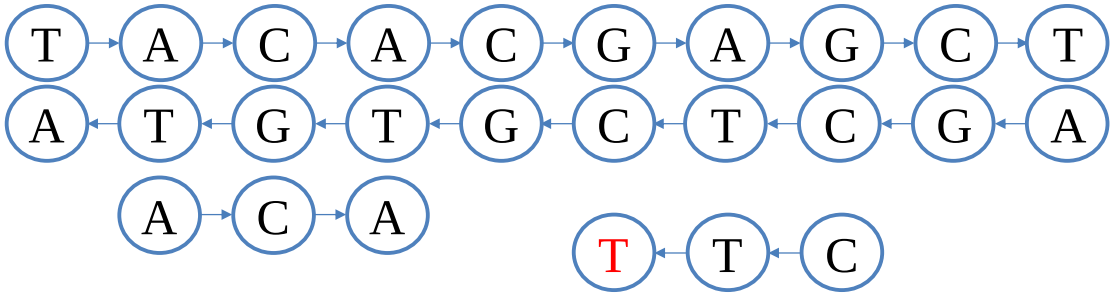

Figure 4.2: A fixed-length single-end read can be recovered by knowing the starting location where the read aligns to the reference, the number of alignment differences, the positions of these differences, and the differing bases. Further, a paired-end read can be recovered from two single-end reads by knowing the location of the other end for one of the ends

In the following sections, we develop fast and scalable parallel algorithms for accomplishing tasks that make up our solution approach. By utilizing these algorithms, we demonstrate that ParRefCom achieves superior compression efficiency compared to existing methods. Our compressor is asymmetric by design - decompression speed is about an order of magnitude faster than compression speed. This is because the decompression step does not involve the computationally expensive specialized alignment phase. This asymmetricity in design goes well with the real world requirement of compressing a dataset once and decompressing (using) it many times. We describe our experimental methodol-

ogy next.

4.5 Methodology

Table 4.1 lists key properties of datasets used for evaluating this work. The datasets correspond to organisms from three different species with varying genome lengths. These datasets are publicly available and were previously used to benchmark many special-purpose compression tools. Column 2 depicts the approximate size of the standard reference genome for each organism. Column 3 represents the number of fixed-length paired-end reads in each dataset. The length of these reads is captured in Column 4 and sizes of the original datasets in Column 5.

Table 4.1: Datasets used for experimental evaluation

Organism	Genome length (Base-pairs)	Spots/Inserts (Thousands)	Read length (Bases)	Original Size	Reference Sequence
C. elegans	100×10^6	33,809	2×100	6.8 GB	GCF_000002985.6
G. gallus	1125×10^6	173,698	2×100	34.7 GB	GCF_000002315.4
H. sapiens(H1)	3300×10^6	24,476	2×100	4.9 GB	GCF_000001405.38
H. sapiens(H2)	3300×10^6	207,680	2×101	42 GB	GCF_000001405.38
H. sapiens(H3)	3300×10^6	270,765	2×146	79 GB	GCF_000001405.38

Accession numbers for C. elegans, H. sapiens (H1) and H. sapiens (H2) are SRR065390, SRR062634, and ERR174310 respectively. The process for obtaining G. gallus dataset is described in [34]. H. sapiens (H3) dataset was generated using Illumina NovaSeq and is available from Illumina BaseSpace as NA12878-Rep-1_S1_L001. This dataset contains variable length reads with length up to 151. We trimmed the reads down to 146 bases to make them fixed-length while retaining the maximum number of reads. We use bsc (`http://libbsc.com`) to compress several streams of information generated in Sections 4.7 and 4.8.

We ran our experiments on a machine with two 14-core Intel Xeon processors, for a total of 28 cores. The 28 cores in a node share 256 GB of main memory. Further, we

used POSIX threads for shared-memory parallel programming. For reporting performance results, each experiment was repeated three times. The wall-times were collected for each run, and the minimum times amongst the repeats of an experiment are reported as they closely represent the capabilities of the system.

4.6 Specialized Alignment

4.6.1 Motivation

The goal of reference-based special-purpose compression algorithms is to represent the read dataset compactly by capturing differences in reads with respect to reference genome. Our specialized compressor, ParRefCom, also employs a reference-based strategy. Aligning reads in a dataset, generated from sequencing a target genome, to standard reference is a fundamental bioinformatics problem. The objective of read alignment is to glean biological insights by computing and analyzing variations between target genome and reference genome.

Some attributes of classical alignment are detrimental to compression efficiency when the objective is to represent a read dataset compactly. When there are multiple equally good alignments for a read, classical alignment tools may report more than one alignment for such reads. For compression, one best alignment suffices. Generating alternative alignments for reads also involves a computational overhead. While aligning paired-end reads, classical alignment tools perform analysis to estimate the insert size and discard *abnormal* alignments [47]. For example, when aligning a paired-end (PE) read, the location of the read whose reverse-complement aligns to the reference genome is expected to be after that of the read that aligns as is. If this is not the case, the alignment may be discarded. As we are not concerned about biological significance during compression, the compression efficiency would improve if such an alignment were to be accepted. Finally, clipping of reads performed by classical alignment tools may impede faithful recovery of such reads.

In this section, we propose a specialized alignment algorithm (SAA) that addresses the

drawbacks of the classical alignment algorithms when the objective of the alignment is to represent a read dataset compactly. The goal of our SAA is as follows. For a PE read, we want to generate an alignment that minimizes the following expression : $D_1 + D_2 + S_{12}$; D_1 : Differences between first read and reference genome, D_2 : Differences between second read and reference genome, and S_{12} : Separation between alignment locations of two reads. We propose an SAA utilizing kmer-index that optimizes the stated objective function. Next, we describe the kmer-index data structure and provide a parallel algorithm for constructing it.

4.6.2 Index Data Structure

Our kmer-index data structure comprises of two levels, each an array, as depicted in Figure 4.3. Level 2 (L2) array consists of locations of all kmers in the reference genome. The entries are sorted by kmer as primary key, and for a kmer by location as secondary key. The size of level 1 (L1) array is $4^k + 1$. Each entry in this array, excluding the last one, corresponds to a kmer and captures the starting position of the corresponding kmer in L2 array. If a kmer does not occur in the reference genome, it does not have any entries corresponding to it in L2 array. For such kmers, the corresponding L1 array entry contains the same value as that of the next kmer. To summarize, two consecutive entries in L1 array help determine the number of occurrences of a kmer in the genome. Values recorded in L2 array in the range defined by the two entries provide the corresponding kmer locations.

Using a two-level index data structure has several advantages. The number of occurrences of a kmer in the reference genome can be determined by accessing two consecutive entries in L1 array. This quick determination can aid in reducing the alignment time. Further, the locations where the kmer occurs in the genome are present in a contiguous segment of L2 array. Scanning these locations therefore benefits from spatial locality of cache and/or memory accesses. Memory consumption of the two-level index data structure is $(4^k + 1) \times 2k + (|\mathcal{G}| - k + 1) \times \lceil \log_2 |\mathcal{G}| \rceil$ bits, where $|\mathcal{G}|$ represents the length of the

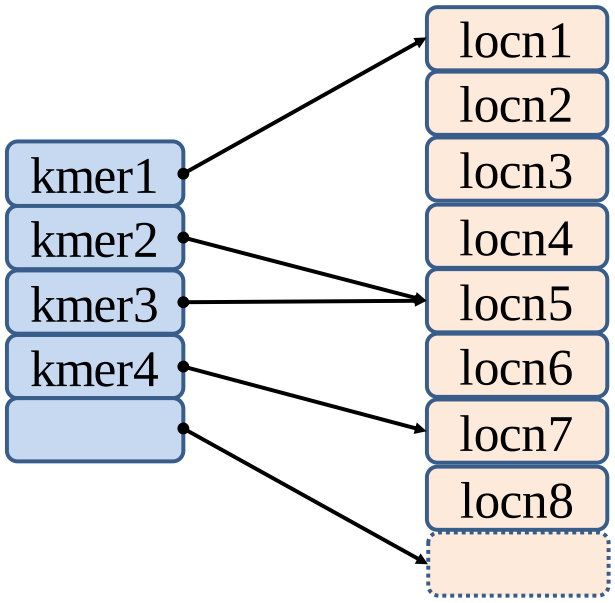

Figure 4.3: An illustration of two-level kmer-index data structure. kmer1, kmer3, and kmer4 occur four, two, and two times respectively in the reference genome. kmer2 does not occur in the reference genome

genome. For large genomes and small values of k (< 15), as used in this work, second term dominates. The overhead due to L1 array is insignificant. Therefore, the two-level index data structure enables fast kmer lookups while incurring a small memory overhead. Next, we present a parallel algorithm for constructing the two-level index data structure.

Parallel Index Construction

Algorithm 10 demonstrates the parallel construction of the two-level index data structure. Initially, reference genome is block decomposed among the available threads. Every thread generates the list of $(kmer, locn)$ tuples for the block owned by the thread. Next, the list across all threads is sorted in parallel. $locn$s from the resulting sorted list make up L2 array. L1 array is obtained by performing a parallel prefix on the sorted list of tuples. Populating L1 array entries corresponding to kmers not occurring in $\mathcal{G}$ is handled as described in Section 4.6.2.

Algorithm 10: Parallel construction of kmer-index

Input: $\mathcal{G}$, reference genome. k, kmer size. t, thread count
Output: Two-level kmer-index: L1 and L2 arrays
// Size of L1 is $4^k + 1$
// Size of L2 is $|\mathcal{G}| - k + 1$. $|\mathcal{G}| \leftarrow$ genome size

parallel *j = thread's id* **do**
  $S_G \leftarrow$ Size of the reference genome $\mathcal{G}$
  $L_G \leftarrow \lceil \frac{S_G}{t} \rceil$
  $lt \leftarrow j \times L_G$
  $rt \leftarrow (j+1) \times L_G$ - 1
  if $(S_G - k + 1) < rt$ **then**
    $rt \leftarrow (S_G - k + 1)$
  end
  Initialize $\mathcal{T}$ to an empty list of tuples
  for $i \leftarrow lt$ **to** rt **do**
    Append $(kmer, locn)$ to $\mathcal{T}$
  end
  Parallel sort $\mathcal{T}$ using $kmer$ as primary key and $locn$ as secondary key
  for $i \leftarrow lt$ **to** rt **do**
    L2[i] $\leftarrow \mathcal{T}[i].locn$
  end
  Use parallel prefix to populate L1 with start position of every kmer in $\mathcal{T}$
end

4.6.3 Alignment Algorithm

The objective of our specialized alignment algorithm (SAA) is as follows. For a paired-end (PE) read, we want to generate an alignment that minimizes the following expression : $D_1 + D_2 + S_{12}$; D_1 : Differences between first read and reference genome, D_2 : Differences between second read and reference genome, and S_{12} : Separation between alignment locations of two reads. In this section, we describe the design of such an SAA and its parallel implementation. In case of a PE read, we assume that one of the ends is sequenced from the forward strand and the other from the reverse-complementary strand. This assumption is representative of the most common form of paired-end sequencing, called forward-reverse sequencing.

Let e denote the number of differences we wish to tolerate between a read and the

reference genome. According to pigeon-hole principle, if the read is decomposed into $(e + 1)$ non-overlapping subsequences, then there is at least one subsequence of the read that matches exactly with an identical length subsequence from the reference genome. This property can be extended as follows. If the read is decomposed into $(e+2)$ non-overlapping subsequences, then there are at least two subsequences of the read that match exactly with identical length subsequences from the reference genome. The extended pigeon-hole principle has the ability to filter out spurious matches more effectively. Note that the extension does not have any impact on the alignment output but only serves to potentially reduce the computational cost. Therefore, we adopt the extended pigeon-hole principle in our SAA.

A high-level overview of our SAA is as follows. First, we decompose a read into non-overlapping kmers. We look up each kmer in L1 array to determine its frequency of occurrence. We select a subset of $(e + 2)$ kmers that occur with the lowest frequency. For these $(e + 2)$ kmers, we look up their corresponding locations in L2 array. Next, we obtain a subset of locations which correspond to at least two kmers. We perform a banded-alignment of the read at the locations in the subset using a vectorized Meyer's bit-vector algorithm. If the read aligns to the reference sequence with $\leq e$ differences, we record the start position of the alignment and the differences.

For a PE read, we perform the above steps for each individual read and its reverse-complement. From this point on, for simplicity, we refer to a read that aligns as is to the reference sequence as forward read and a read whose reverse-complement aligns to the reference sequence as reverse read. We then select the best alignment as one that minimizes the sum of (1) Differences between forward read and reference genome, (2) Differences between reverse read and reference genome, and (3) Separation between alignment start locations of the two reads. Further, we classify PE reads based on the outcome of the alignment as : Two-aligned (when both ends of the PE read align), One-aligned (when one of the ends of the PE read aligns but the other does not), and Non-aligned (when both ends of the PE read do not align). We present our strategies for handling reads in each

of the three categories in subsequent sections. The parallel implementation of our SAA is described in Algorithm 11.

SAA lends itself very well to parallelization. PE read dataset $\mathcal{R}$ is decomposed into virtual blocks, each of size $\mathcal{B}$ reads. Each thread parses $\mathcal{B}$ reads, performs alignment for them, and generates the corresponding output. The output blocks are appended to appropriate lists as described in Algorithm 11. Threads only need to synchronize to determine the blocks of reads to work on. Each thread seeks ownership of a new block of reads once it is done working on the block it currently owns. The parameter $\mathcal{B}$ can be tuned appropriately based on the number of threads to ensure none of the threads starves. Note that a straightforward block decomposition of reads among threads can lead to a load imbalance among threads. This is because the computational effort necessary to align reads varies from one read to another.

4.6.4 Results and Analysis

In this section, we present results obtained using our specialized alignment algorithm (SAA). Before proceeding to discuss the results, we furnish the values used for various parameters and the rationale behind selecting the specific values. For the number of hardware threads typical among shared-memory machines, a value of 2000 for $\mathcal{B}$, read block size, ensures that threads spend almost all their time performing useful work, namely read alignment.

A value of 14 for k allows us to tolerate up to 5 differences per 100 bases ($\lfloor \frac{100}{(5+2)} \rfloor$) between a read and the corresponding subsequence of the reference genome. Note that SAA makes use of extended pigeon-hole principle. We choose a small value of 3 for e initially, to reduce the computational overhead. However, our implementation of the banded and vectorized Meyer's bit-vector algorithm allows a band size of up to 7. We utilize the full potential of the alignment algorithm to allow up to 7 differences. So, the final value of e supported by SAA is 7. Finally, for every read, we explore up to 2000

Algorithm 11: Parallel specialized alignment algorithm

Input: $\mathcal{R}$, paired-end read dataset. k, kmer size. t, thread count
Input: $\mathcal{B}$, read block size. e, number of differences
Output: LF_{Two}: List of forward two-aligned reads
Output: LR_{Two}: List of reverse two-aligned reads
Output: L_{One} : List of paired-end one-aligned reads
Output: L_{Non} : List of paired-end non-aligned reads
// Reads in LF_{Two} and LR_{Two} correspond one to one

```
parallel j = thread's id do
    while reads in R do
        Parse B reads from R
        for each r in B do
            r1 ← first read in r
            r2 ← second read in r
            Align r1 and r2 as described in Section 4.6.3
            if r1 and r2 align then
                rf ← forward read. rr ← reverse read
                Append (rf, locnf, id) to LF_Two
                Append (rr, locnr, id) to LR_Two
            else if r1 or r2 aligns then
                if r1 aligns as forward read then
                    rf ← r1. rr ← reverse of r2
                    Append (rf, rr, locn1) to L_One
                else if r1 aligns as reverse read then
                    rf ← r2. rr ← reverse of r1
                    Append (rf, rr, locn1) to L_One
                else if r2 aligns as forward read then
                    rf ← r2. rr ← reverse of r1
                    Append (rf, rr, locn2) to L_One
                else
                    rf ← r1. rr ← reverse of r2
                    Append (rf, rr, locn2) to L_One
                end
            else
                Determine rf, rr, and locn as described in Section 4.8
                Append (rf, rr, locn) to L_Non
            end
        end
    end
end
```

potential locations to select the best alignment. Note that the ability to align a read, on average, tapers off as the number of explored locations increases. The chosen value helps achieve a good trade off between the number of reads aligned and the computational cost incurred.

Table 4.2 shows the percentage of paired-end (PE) reads falling into each of the three categories : two-aligned, one-aligned, and non-aligned, for all datasets using SAA. It can be observed that the percentage of two-aligned reads is the highest for all datasets, with the value for human dataset reaching 90%. PE reads in this category offer the most potential for compression. We present our strategy for compressing reads in two-aligned category in Section 4.7. It should be noted that *C. elegans* and *G. gallus* datasets were generated using older generation of sequencing instruments. Therefore, these datasets contain more sequencing errors, which are partly responsible for the lower percentage of PE reads in two-aligned category compared to human datasets. As we mentioned previously, these results correspond to the case where we tolerate up to 7 differences per single-end read. Our strategy for compressing the reads falling into one-aligned and non-aligned categories is described in Section 4.8.

Table 4.2: Percentage of paired-end reads aligned using specialized alignment algorithm

Dataset	Two-aligned	One-aligned	Non-aligned
C. elegans	85.12	8.17	6.71
G. gallus	84.14	5.87	9.99
H. sapiens(H1)	88.57	8.21	3.22
H. sapiens(H2)	90.04	7.94	2.02
H. sapiens(H3)	88.46	8.84	2.70

In Section 4.6.1, we described the rationale behind designing a specialized alignment algorithm to complement classical alignment algorithms when the objective is to represent the read dataset in a compact manner. Table 4.3 shows the percentage of PE reads falling into various categories for all datasets using BWA, a flagship classical alignment tool. It can be observed that the output of BWA comprises of two additional categories : clipped

and multi-aligned. For PE reads in the clipped category, we do not have complete alignment for at least one of the ends. In case of multi-aligned category, we have multiple alignments for at least one of the ends. Even thought the percentage of two-aligned PE reads is higher for some datasets, the reads in clipped and multi-aligned categories pose challenges for compression. BWA was run with default parameters and tolerates more differences per single-end read than SAA, which is mostly responsible for the higher percentage of two-aligned PE reads.

Table 4.3: Percentage of paired-end reads aligned using BWA

Dataset	Two-aligned	One-aligned	Non-aligned	Clipped	Multi-aligned
C. elegans	81.63	0.65	3.85	12.71	1.16
G. gallus	80.07	0.24	0.45	18.71	0.53
H. sapiens(H1)	89.88	0.23	0.16	9.27	0.46
H. sapiens(H2)	93.94	0.23	0.42	5.06	0.35
H. sapiens(H3)	94.26	0.09	0.31	4.58	0.76

Table 4.4 shows the time taken by SAA and BWA for alignment using identical number of threads, 16 in this case. It can be noticed that SAA is nearly an order of magnitude faster than BWA. This demonstrates the superior computational capability of SAA compared to classical alignment tools for the purpose of representing read datasets compactly. Table 4.5 shows the runtime incurred by SAA for human dataset H1 as the number of threads is increased from 2 to 16. It can be observed that SAA demonstrates good parallel scalability. In the following section, we describe our algorithm and its parallel implementation for handling PE reads in the two-aligned category.

Table 4.4: Alignment time in seconds using 16 threads for specialized alignment algorithm and BWA

Dataset	SAA time (s)	BWA time (s)	SAA speedup
C. elegans	74	619	8.37
G. gallus	717	7898	11.02
H. sapiens(H1)	161	1383	8.59
H. sapiens(H2)	1379	12210	8.85
H. sapiens(H3)	2780	20387	7.33

Table 4.5: Runtime in seconds for *H. sapiens* (H1) dataset using specialized alignment algorithm

No. of threads	Time (s)
2	940
4	473
8	249
16	133

4.7 Handling Two-aligned Paired-end Reads

4.7.1 Overall Approach

For two-aligned paired-end (PE) reads, we capture the following pieces of information in lieu of the reads themselves : (1) Alignment start location with respect to the reference sequence, (2) Number of differences with respect to the reference sequence, (3) Positions of differences within a read, (4) Bases corresponding to differences within a read and (5) Locations of other ends for one of the ends. Collectively, these pieces of information are sufficient to reconstruct two-aligned PE reads. In the following subsections, we provide a detailed description for generating each of these pieces of information. Next, we present our algorithm for generating the above pieces of information.

Algorithm 12 demonstrates parallel generation of data streams to be recorded for two-aligned reads. Lists of forward and reverse two-aligned reads generated by Algorithm 11 are independently sorted in parallel based on the alignment locations of the reads. The sorted lists are block decomposed among available threads and each thread is responsible for generating data streams for reads in the block owned by it. Data stream (5) is an exception and we describe the process for generating it in Section 4.7.6. We elaborate on the generation of data streams (1)-(4) in the following subsections. Note that data streams (1)-(4) are generated for both forward and reverse reads and data stream (5) only for forward reads. This is because it is sufficient to record pairing information only for one of the ends, forward read in our case, of a PE read. The generated data streams are compressed

in parallel using bsc, a general-purpose compressor.

Algorithm 12: Parallel generation of data streams for two-aligned reads

Input: LF_{Two}: List of forward two-aligned reads
Input: LR_{Two}: List of reverse two-aligned reads
Output: LF_{SL} and LR_{SL}: List of starting locations
Output: LF_{ND} and LR_{ND}: List of number of differences
Output: LF_{PD} and LR_{PD}: List of positions of differences
Output: LF_{BD} and LR_{BD}: List of bases corresponding to differences
Output: L_{PE}: List of locations of other ends
// LF_{Two} and LR_{Two} have the same size
// They are lists of tuples of type $(read, locn, id)$
1 **parallel** *j = thread's id* **do**
2 $S_G \leftarrow$ Size of LF_{Two}. $L_G \leftarrow \lceil \frac{S_G}{t} \rceil$
3 $lt \leftarrow j \times L_G$. $rt \leftarrow (j+1) \times L_G$ - 1
4 **if** $(S_G - 1) < rt$ **then**
5 $rt \leftarrow (S_G - 1)$
6 **end**
7 Parallel sort LF_{Two} using $locn$ as key. Parallel sort LR_{Two} using $locn$ as key
8 Initialize $\mathcal{T}F$ to an empty list of tuples. Initialize $\mathcal{T}R$ to an empty list of tuples
9 **for** $i \leftarrow lt$ **to** rt **do**
10 Append $(idf, locnf, posnf)$ to $\mathcal{T}F$
11 Append $(idr, locnr)$ to $\mathcal{T}R$
12 Append starting location of $LF_{Two}[i]$ to LF_{SL}
13 Append starting location of $LR_{Two}[i]$ to LR_{SL}
14 Append number of differences of $LF_{Two}[i]$ to LF_{ND}
15 Append number of differences of $LR_{Two}[i]$ to LR_{ND}
16 Append positions of differences of $LF_{Two}[i]$ to LF_{PD}
17 Append positions of differences of $LR_{Two}[i]$ to LR_{PD}
18 Append differing bases of $LF_{Two}[i]$ to LF_{BD}
19 Append differing bases of $LR_{Two}[i]$ to LR_{BD}
20 **end**
21 Parallel sort $\mathcal{T}F$ using id as key. Parallel sort $\mathcal{T}R$ using id as key
22 Initialize $\mathcal{T}P$ to an empty list of tuples
23 **for** $i \leftarrow lt$ **to** rt **do**
24 Append $(locnf, locnr, posnf)$ to $\mathcal{T}P$
25 **end**
26 Parallel sort $\mathcal{T}P$ using $locnr$ as key
27 **for** $i \leftarrow lt$ **to** rt **do**
28 Append $(locnr - loncnf, posnf)$ to L_{PE}
29 **end**
30 **end**

4.7.2 Generating List of Starting Locations

The first piece of information that needs to be recorded for every read, forward and reverse, is the location in the reference genome where the read starts aligning with the genome. There are two possible options for recording the alignment start location information : (1) Combine forward and reverse reads into a single list or (2) Maintain them in separate lists. In the former case, in addition to storing the start location, a bit is necessary to indicate whether the read is a forward read or a reverse read. Further, there can be implication for how the pairing information is maintained. Owing to these reasons, it is beneficial to choose the second option. For a read, we store its relative start location, location delta from previous read, instead of the absolute start location. This offers addition space savings.

For all datasets, and for both forward and reverse lists, the frequency of occurrence of relative start location values falls off sharply as the values increase. Based on this observation, we use the following scheme to encode location deltas. We use one byte to encode the location delta if the value is < 253. We use the remaining three values that can be represented using a byte - $253, 254,$ and 255 - to indicate that $2, 3,$ and 4 bytes are necessary to capture the location delta value respectively. The value is stored using the corresponding number of bytes.

4.7.3 Generating List of Number of Differences

The next piece of information that needs to be recorded for every read is the number of bases different in the alignment between the read and the reference genome. We use one byte to capture each difference count.

4.7.4 Generating List of Positions of Differences

In addition to storing the counts of differences, we need to record the positions of differing bases for reads which have one or more differences. The number of bits necessary to store each difference is $log_2 L$, where L denotes the length of the read. This cost can be brought

down if we store relative difference positions. We use one byte to encode each relative difference position.

4.7.5 Generating list of Bases Corresponding to Differences

In order to recreate individual reads exactly during decompression, we need to record the bases corresponding to the differences along with the counts and the positions of the differences. In our experiments, we observed that the count of substitutions significantly outnumber the counts of insertions and deletions. Further, the count of insertions and deletions is approximately the same.

Taking these characteristics into account, we developed the following encoding scheme to capture the base differences. We use 2 bits for capturing a substitution. Note that $\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}$ can be substituted with one of the other three bases. This leaves us with one unused value, 11, which can be used to capture additional scenarios using 2 more bits. The four values made available through the additional 2 bits are used to capture : substitution to an `N`, deletion, insertion of an `N`, and insertion of a regular base. When the insertion corresponds to a regular base, we use 2 additional bits to capture the inserted base.

4.7.6 Generating List of Locations of Other Ends

Collectively, data streams (1) - (4) described in preceding subsections contain sufficient information to recover forward and reverse reads independently. We now describe the information we capture so that reverse reads can be paired with their corresponding forward reads. Among the approaches available to capture pairing information, the one that incurs the least cost is the following. For every reverse read, we capture the relative location of its forward read. There can be more than one forward read that aligns starting at a given location. To account for such cases, we capture which of these forward reads pairs with the reverse read under consideration.

We use one byte to encode each value of the second type. The former values follow a

normal distribution. We map the distribution mean to zero and compute the corresponding folded-normal distribution. We use the scheme described in Section 4.7.2 to encode the resulting values. This concludes our discussion on capturing the data streams necessary for encoding and decoding two-aligned PE reads. Next, we discuss our approach for handling one-aligned and non-aligned PE reads.

4.8 Handling One- and Non- Aligned Paired-end Reads

In Section 4.6.4, we showed that there are a small fraction of paired-end (PE) reads for which one or both ends do not align. We classified them under one-aligned and non-aligned categories, respectively. Even though we do not have the desirable outcome for these PE reads, our specialized alignment algorithm (SAA) generates sufficient information to orient and order them. In order to orient PE reads, we need to know which read to capture as forward read and which one to capture as reverse read. Relative placement of PE reads with respect to one another helps in determining a good ordering to assist in better compression. In case of one-aligned PE reads, we have an alignment for one of the ends of the PE read. Based on whether the end is aligned as a forward read or as a reverse read, we have the necessary information to determine the orientation. Further, location of the aligned end assists in ordering the reads relative to one another.

The determination of orientation and ordering of non-aligned PE reads works as follows. Note that we don't have an alignment for any of the ends in case of a non-aligned read. When we attempt to align a PE read using SAA, we generate potential candidate locations where the read may align. When such candidate locations are available for a non-aligned PE read, we pick the best among available locations and use it to orient and order reads. We label such PE reads under non-aligned-x category. Even when potential candidate locations are not available, we can have one or more successful kmer lookups. When information from looking up kmers is available, we pick the best among such lookups and use it to orient and order reads. We label such PE reads under non-aligned-y category. Fi-

nally, when we have no information available to orient and order PE reads, we label such reads under non-aligned-z category.

The orientation for PE reads in non-aligned-z category is determined based on the order in which reads appear in the dataset, first read as forward read and second read as reverse read. In terms of ordering PE reads in this category, we place them after PE reads belonging to the remaining categories. In Section 4.10, we will show that the percentage of reads in non-aligned-z category is zero or close to it for all datasets. Therefore, even when both ends of a PE read do not align, we have sufficient evidence to orient and order reads.

For one- and non-aligned PE reads, orientation is determined while attempting to align the reads using SAA. Further, they are ordered by performing a parallel sort. We use one byte per base encoding to capture one- and non- aligned PE reads. Oriented and ordered one- and non- aligned PE reads are compressed in parallel using bsc, a general purpose compressor. We considered alternative encodings, including using two bits for four regular bases and recording Ns separately, but one byte per base encoding yielded the best compression efficiency. It must be noted that classical alignment algorithms do not provide any information for orienting and ordering non-aligned reads.

4.9 Decompression Algorithm

Our special-purpose compressor, ParRefCom, is asymmetric by design - decompression is about an order of magnitude faster than compression. This is because the decompression step does not involve the expensive specialized alignment phase. This asymmetric design of ParRefCom goes well with the real world requirement of compressing a dataset once and decompressing (using) it many times. Our parallel algorithm for recovering the read dataset using the compressed representations described in Sections 4.7 and 4.8 works as described below. General-purpose compressor bsc is first used to perform parallel decompression. Recovery of one- and non- aligned PE reads is complete after this step. Note that reverse-complement of second end in every PE read is computed to undo the reverse-complement

operation performed by the specialized alignment algorithm.

4.9.1 Recovering Two-aligned Reads

Our parallel algorithm for recovering two-aligned PE reads is described in Algorithm 13. We first use the reference genome and the list of starting locations to recover difference-free reads. Then, we use lists of counts of differences, positions of differences, and bases corresponding to differences in order to update bases that are different between reads and the reference genome. These steps are performed independently for lists corresponding to forward and reverse reads. Finally, we perform a parallel sort of reverse reads using information contained in data stream (5). This step accomplishes the task of pairing reverse reads with forward reads appropriately.

4.10 Results and Analysis

In this section, we present and analyze results corresponding to compression and decompression of all datasets. We use SPRING, the current best compression algorithm, as baseline to compare our results. Table 4.6 shows the sizes of the compressed datasets, in mega bytes, produced by SPRING and ParRefCom. It can be observed that ParRefCom performs better than SPRING for all datasets but *C. elegans*. Excluding the *C. elegans* dataset, the size of the compressed output produced by ParRefCom is at least 21% smaller compared to that produced by SPRING. For human dataset H1, it is smaller by as much as 77%. These results demonstrate the superior compression efficiency of ParRefCom, our special-purpose compressor.

Tables 4.7 and 4.8 depict the time taken by SPRING and ParRefCom tools for compression and decompression respectively. The results are provided for all datasets and correspond to the case when 16 threads are used. For compression, ParRefCom is at least 1.5 times faster than SPRING across all datasets. In case of decompression, the performance gain is even more prominent, and ParRefCom is at least 1.7 times faster than SPRING

Algorithm 13: Parallel algorithm for recovering two-aligned reads

Input: $\mathcal{G}$, reference genome. t, thread count
Input: C, number of two-aligned paired-end reads
Input: LF_{SL} and LR_{SL}: List of starting locations
Input: LF_{ND} and LR_{ND}: List of number of differences
Input: LF_{PD} and LR_{PD}: List of positions of differences
Input: LF_{BD} and LR_{BD}: List of bases corresponding to differences
Input: L_{PE}: List of locations of other ends
Output: LF_{Two}: List of forward two-aligned reads
Output: LR_{Two}: List of reverse two-aligned reads
// Reads in LF_{Two} and LR_{Two} correspond one to one
parallel *j = thread's id* **do**
  $S_G \leftarrow$ Size of LF_{Two}
  $L_G \leftarrow \lceil \frac{S_G}{t} \rceil$
  $lt \leftarrow j \times L_G$
  $rt \leftarrow (j+1) \times L_G$ - 1
  if $(S_G - 1) < rt$ **then**
    $rt \leftarrow (S_G - 1)$
  end
  for $i \leftarrow lt$ **to** rt **do**
    Populate $LF_{Two}[i]$ using decoded $LF_{SL}[i]$ and $\mathcal{G}$
    Populate $LR_{Two}[i]$ using decoded $LR_{SL}[i]$ and $\mathcal{G}$
    Identify differing bases in $LF_{Two}[i]$ using $LF_{ND}[i]$,
    and corresponding number of LF_{PD} entries
    Identify differing bases in $LR_{Two}[i]$ using $LR_{ND}[i]$,
    and corresponding number of LR_{PD} entries
    Update differing bases in $LF_{Two}[i]$ using $LF_{ND}[i]$,
    and corresponding number of decoded LF_{BD} entries
    Update differing bases in $LR_{Two}[i]$ using $LR_{ND}[i]$,
    and corresponding number of decoded LR_{BD} entries
    Compute reverse-complement of $LR_{Two}[i]$ read
  end
  Sort LR_{Two} in parallel using L_{PE} values as keys
  // $LF_{Two}[i]$ and $LR_{Two}[i]$ correspond to two ends of a paired-end read
end

across all datasets. Note that a dataset typically needs to be compressed only once but needs to be decompressed multiple times. Therefore, decompression performance carries more significance than compression performance.

Table 4.9 shows the percentage of reads corresponding to each difference value for all

Table 4.6: Compression efficiency of SPRING and ParRefCom algorithms

Dataset	SPRING size (MB)	ParRefCom size (MB)	ParRefCom improvement
C. elegans	227	267	-17.62%
G. gallus	1,512	1,174	22.36%
H. sapiens(H1)	825	192	76.72%
H. sapiens(H2)	1,666	1,322	20.65%
H. sapiens(H3)	2,022	1,457	27.94%

Table 4.7: Compression time in seconds using 16 threads for SPRING and ParRefCom algorithms

Dataset	SPRING time (s)	ParRefCom time (s)	ParRefCom speedup
C. elegans	398	122	3.26
G. gallus	2,343	1,041	2.25
H. sapiens(H1)	557	364	1.53
H. sapiens(H2)	2,817	1,789	1.58
H. sapiens(H3)	4,890	3,301	1.48

datasets. It can be observed that the percentage of reads with no differences is the highest for all datasets and the percentage of reads with differences falls sharply as the difference value increases. Further, the percentage of reads with no differences and the percentage of reads with one difference together account for about 90% of total reads for all datasets.

Table 4.10 shows the distribution of various types of differences - namely substitutions, insertions, and deletions - for all datasets. It can be observed that the percentage of substitutions significantly outnumber the percentages of insertions and deletions. Further, the percentages of insertions and deletions are approximately the same, particularly for human datasets.

Table 4.11 shows the breakdown for one- and non- aligned PE reads in each of the categories described in Section 4.8 for all datasets. It can be observed that the percentage of reads in non-aligned-z category is zero or close to it for all datasets. Therefore, even when both ends of a paired-end read do not align, we have sufficient evidence to orient and order reads. Note that classical alignment algorithms do not provide any information for orienting and ordering non-aligned reads.

Table 4.8: Decompression time in seconds using 16 threads for SPRING and ParRefCom algorithms

Dataset	SPRING time (s)	ParRefCom time (s)	ParRefCom speedup
C. elegans	36	20	1.80
G. gallus	227	133	1.71
H. sapiens(H1)	69	32	2.16
H. sapiens(H2)	264	133	1.99
H. sapiens(H3)	415	203	2.04

Table 4.9: Counts of differences (percentage) for two-aligned paired-end reads using the specialized alignment algorithm

Differences	*C. elegans*	*G. gallus*	*H. sapiens*(H1)	*H. sapiens*(H2)	*H. sapiens*(H3)
0	79.06	77.92	71.45	76.92	74.42
1	10.74	10.67	17.22	13.48	15.74
2	3.68	3.65	4.70	3.80	4.43
3	2.25	2.25	2.24	1.91	2.02
4	1.59	1.75	1.48	1.31	1.29
5	1.16	1.49	1.12	0.98	0.89
6	0.87	1.26	0.95	0.83	0.67
7	0.66	1.01	0.85	0.77	0.55

4.11 Discussion

4.11.1 Compression Metadata

In addition to the streams of data generated to represent two-aligned, one-aligned, and non-aligned paired-end (PE) reads compactly, we need to capture the following metadata in order to recover the read dataset : (1) Number of threads, (2) Identifier for reference genome, (3) Read length, (4) Sizes of lists generated in Sections 4.7 and 4.8, and (5) Information corresponding to the transformation performed before encoding described in Section 4.7.6. The storage cost incurred by metadata is less than 1 KB, and is insignificant in comparison to that incurred to represent the compressed output.

Table 4.10: Types of differences (percentage) for two-aligned paired-end reads using the specialized alignment algorithm

Difference type	*C. elegans*	*G. gallus*	*H. sapiens*(H1)	*H. sapiens*(H2)	*H. sapiens*(H3)
Substitution	94.49	90.11	94.13	93.80	91.55
Insertion	3.52	6.30	3.03	3.03	3.97
Deletion	1.99	3.59	2.84	3.17	4.48

Table 4.11: Percentage of one- and non- aligned PE reads using the specialized alignment algorithm

Difference type	*C. elegans*	*G. gallus*	*H. sapiens*(H1)	*H. sapiens*(H2)	*H. sapiens*(H3)
One-aligned	8.17	5.87	8.21	7.94	8.84
Non-aligned-x	2.74	9.09	2.96	1.51	2.34
Non-aligned-y	3.95	0.98	0.26	0.51	0.36
Non-aligned-z	0.02	0.00	0.00	0.00	0.00

4.11.2 Verifying Decompressed Output

Recall that our specialized compressor, ParRefCom, allows the following lossless transformations of PE reads in the datasets : reordering of reads while keeping paired ends together and reordering of individual reads within a PE read. Therefore, the decompressed output has the same information content as, but is not exactly identical to the input read dataset. We propose a parallel algorithm to verify that the decompressed output contains the same information as the input, while allowing the two lossless transformations described previously. It is described in Algorithm 14.

Algorithm 14: Parallel algorithm for verifying decompressed output

Input: $\mathcal{R}$, input paired-end read dataset. t, thread count
Input: $\mathcal{R}'$, paired-end read dataset recovered after decompression
Output: TRUE, if information content in $\mathcal{R}$ and $\mathcal{R}'$ is same. Else, FALSE

parallel *j = thread's id* **do**
  if $|\mathcal{R}| != |\mathcal{R}'|$ **then**
    return FLASE
  end
  $S_G \leftarrow |\mathcal{R}|$
  $L_G \leftarrow \lceil \frac{S_G}{t} \rceil$
  $lt \leftarrow j \times L_G$
  $rt \leftarrow (j+1) \times L_G$ - 1
  if $(S_G - 1) < rt$ **then**
    $rt \leftarrow (S_G - 1)$
  end
  Initialize $\mathcal{T}$ to an empty list of tuples
  Initialize $\mathcal{T}'$ to an empty list of tuples
  for $i \leftarrow lt$ **to** rt **do**
    Sort two ends of $\mathcal{R}[i]$ in lexicographic order and
    append (r_1, r_2) tuple to $\mathcal{T}$
    Sort two ends of $\mathcal{R}'[i]$ in lexicographic order and
    append (r'_1, r'_2) tuple to $\mathcal{T}'$
  end
  Parallel sort $\mathcal{T}$ in lexicographic order using r_1 as primary key
  and r_2 as secondary key
  Parallel sort $\mathcal{T}'$ in lexicographic order using r'_1 as primary key
  and r'_2 as secondary key
  for $i \leftarrow lt$ **to** rt **do**
    if $\mathcal{T}[i].r_1 != \mathcal{T}'[i].r'_1$ **then**
      return FALSE
    end
    if $\mathcal{T}[i].r_2 != \mathcal{T}'[i].r'_2$ **then**
      return FALSE
    end
  end
  return TRUE
end
// Global output is computed as logical AND of local outputs

CHAPTER 5
CONCLUSIONS AND FUTURE WORK

In this chapter, we summarize the contributions of this thesis for solving the following three problems related to high-throughput sequencing short read datasets : (1) error correction, (2) partitioning, and (3) compression. Further, we outline avenues for extending the solutions described in this thesis.

5.1 Error Correction

Genomic read error correction improves the quality of results produced by applications in areas such as genomics, metagenomics, and transcriptomics. Further, it leads to a reduction in the runtime and the memory usage of such applications. Serial error correction methods cannot handle the large number of reads sequenced by modern instruments. A distributed-memory Parallel Spectrum-based Error Correction (PSbEC) algorithm was proposed to address this shortcoming [1]. The PSbEC algorithm suffers from three major problems: (1) A separate copy of the spectrum is maintained per process. This approach does not scale to billion base long genomes, (2) Work is statically allocated to processes. Due to differences in distribution of errors among reads, this leads to significant load imbalance among processes, and (3) Error correction involves repeated binary searches over the spectrum. Binary search is inefficient in terms of memory accesses given the memory organization of modern day computer systems.

In this work, we proposed the following techniques to address the above shortcomings of the PSbEC algorithm: (1) Save only one copy of the spectrum per physical node instead of a copy of the spectrum per process, (2) A dynamic work allocation scheme to solve the load imbalance problem, and (3) A cache-aware layout to represent the spectrum in order to improve the memory-access efficiency. Our proposed strategies enhance the scope and

the speedup of the PSbEC algorithm to accomplish read error correction of big genomic datasets. More specifically, by combining our optimizations, we achieved a cumulative speedup of up to 11 X. In addition, we demonstrated distributed-memory error correction of a dataset consisting of nearly 1.55 billion reads for the first time. We also presented a parallel algorithm for constructing the cache-aware layout of the spectrum.

While developing parallel algorithms, significant emphasis is placed on reducing communication as it is considered to be expensive. In a similar manner, importance needs to be given to optimizing memory-accesses while developing algorithms. This is especially important in case of modern systems in which a memory access is about a few hundred times more expensive than the cost of computation. In this spirit, we adopted the cache-aware layout to improve the performance of the PSbEC algorithm. We hope that our work will spur a renewed interest in embracing such memory-access efficient data structures and algorithms for various other problems.

During the course of our work, we realized that the spectrum can be represented more compactly using a lossless compression scheme. Moreover, searches can be performed more rapidly over the compacted spectrum representation. Representing the spectrum more compactly will enable error correction of bigger datasets using a fixed amount of main memory. Developing the compression scheme and analyzing the corresponding benefits can serve as interesting opportunities for future research. In our approach, the cost of querying an element in the spectrum is $O(logN)$, where N is the number of elements in the spectrum. Alternatively, the spectrum can be stored in the form of a hash table. This approach reduces the cost of querying an element in the spectrum but potentially increases the memory footprint of the spectrum. A potential opportunity for future research is to analyze the time-space trade off involved and select the approach that works better for a given scenario. Hamming graph has applications in other problems associated with read datasets and the algorithms proposed in this work can be adapted to develop solutions for such problems.

5.2 Read Partitioning

Analysis of large-scale datasets produced by modern day high-throughput sequencing instruments poses significant computational and memory challenges. The attempt to address the memory aspect of the challenge using shared-memory machines has limited effectiveness due to the limitations on available memory and number of hardware threads available. To address these challenges, we proposed locality-sensitive partitioning of read sets that can be applied for communication-efficient distributed memory parallel analysis for many applications. Our solution facilitates scalable analysis of the partitioned read datasets in distributed-memory settings, thereby addressing the dual challenges of computation and memory.

As part of generating a partitioning of the read dataset, we construct and partition the de Bruijn graph corresponding to the read set. Partitions of de Bruijn graph can be directly used by applications that predominantly use a variant of the de Bruijn graph. We demonstrate that our proposed solution produces high quality de Bruijn graph and read set partitions for large-scale datasets. The partitioning algorithm is fast, scalable, and effective. Although demonstrated in the context of genomics datasets, it can be extended to applications in areas such as metagenomics and transcriptomics.

Potential opportunities for future research with respect to the read dataset partitioning problem are : Extension of the proposed solution approach to applications in metagenomics and transcriptomics areas. Evaluation of de Bruijn graph and read set partitioning algorithms on additional large-scale datasets. Demonstration of the utility of the proposed partitioning algorithms in the context of actual analyses, based on both de Bruijn graph and read dataset. Utilization of auxiliary information such as reference sequence to improve the quality of partitioning.

5.3 Read Compression

Owing to the large size of the read datasets produced by high-throughput sequencing instruments, transmission, storage, and archival of such datasets pose significant challenges. The magnitude of the problem grows as the sequencing technology improves. Domain-specific properties of read datasets make it difficult for general-purpose compressors to fully exploit the redundancy present in these datasets. Researchers proposed special-purpose compression algorithms, that exploit one or more of the unique properties of read datasets, to improve upon the compression efficiency of general-purpose compressors.

In this work, we developed a reference-based compressor for genomics read datasets which exploits all of the domain-specific properties of read datasets. Our compressor performs a specialized alignment of paired-end (PE) reads to standard reference sequence. It categorizes PE reads based on the number of ends aligned. Finally, it uses custom compression strategies for reads in different categories. Our special-purpose compressor allows lossless transformations of PE reads in datasets. By leveraging all of these insights, it is able to significantly improve upon the compression efficiency over state-of-the-art. In addition to enhancing compression efficiency, we furnished fast and scalable parallel algorithms for compressing and decompressing read datasets.

Currently, our special-purpose compressor only handles *data* portion of read datasets. Integrating it with compression algorithms specifically deigned to handle *metadata* portion of read datasets, to create a unified compressor, is a potential opportunity for future research. Improving the compression efficiency of PE reads, for which one or both ends do not align, can also serve as an interesting avenue for future research.

In our approach, a piece of information that needs to be recorded for every two-aligned PE read is the number of bases different in the alignment between the read and the reference genome. A straightforward approach is to record the count of differing bases as they are. However, there can be variations between the reference genome to which the read dataset

is aligned and the target genome from which the read dataset is generated. These variations manifest themselves in all reads aligning to corresponding locations. It would be redundant to record a variation in every read separately. Note that the manner in which variations are recorded has implication not only for storing the number of differences, but also for storing the positions of differences and the bases corresponding to the differences.

A way to avoid storing a variation redundantly in multiple reads is as follows. As we parse alignments of reads ordered by their alignment locations, we can dynamically update the reference sequence based on differences observed in reads. Using this approach, base at a location in the genome gets updated to the consensus of the reads aligning at that location. So, when a variation appears in more than two reads at a location, the variation is explicitly recorded only for the first two reads. The base gets updated to the variation for the remaining reads, and they don't need to account for the variation explicitly. The process of updating the base is continuous and can be replicated while decompressing the reads as well. The process of updating the base to the consensus makes use of forward and reverse lists simultaneously. Furthermore, it can be implemented using a circular buffer of size $3 \times L$, where L is the length of the read. Implementing this optimization can further improve the compression efficiency for two-aligned PE reads.

REFERENCES

[1] A. R. Shah, S. Chockalingam, and S. Aluru, "A parallel algorithm for spectrum-based short read error correction," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, IEEE, 2012, pp. 60–70.

[2] J. A. Reuter, D. V. Spacek, and M. P. Snyder, "High-throughput sequencing technologies," *Molecular cell*, vol. 58, no. 4, pp. 586–597, 2015.

[3] S. Goodwin, J. D. McPherson, and W. R. McCombie, "Coming of age: Ten years of next-generation sequencing technologies," *Nature Reviews Genetics*, vol. 17, no. 6, pp. 333–351, 2016.

[4] N. Jammula, S. Chockalingam, and S. Aluru, "Parallel read error correction for big genomic datasets," in *IEEE 22nd International Conference on High Performance Computing (HiPC)*, IEEE, 2015, pp. 446–455.

[5] N. Jammula, S. P. Chockalingam, and S. Aluru, "Distributed memory partitioning of high-throughput sequencing datasets for enabling parallel genomics analyses," in *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, ACM, 2017, pp. 417–424.

[6] K. Nakamura, T. Oshima, T. Morimoto, *et al.*, "Sequence-specific error profile of illumina sequencers," *Nucleic acids research*, gkr344, 2011.

[7] N. J. Loman, R. V. Misra, T. J. Dallman, *et al.*, "Performance comparison of benchtop high-throughput sequencing platforms," *Nature biotechnology*, vol. 30, no. 5, pp. 434–439, 2012.

[8] Y. Liu, B. Schmidt, and D. L. Maskell, "Decgpu: Distributed error correction on massively parallel graphics processing units using cuda and mpi," *BMC bioinformatics*, vol. 12, no. 1, p. 85, 2011.

[9] P. Muir, S. Li, S. Lou, D. Wang, D. J. Spakowicz, L. Salichos, J. Zhang, G. M. Weinstock, F. Isaacs, J. Rozowsky, *et al.*, "The real cost of sequencing: Scaling computation to keep pace with data generation," *Genome biology*, vol. 17, no. 1, p. 53, 2016.

[10] A. Sboner, X. J. Mu, D. Greenbaum, R. K. Auerbach, and M. B. Gerstein, "The real cost of sequencing: Higher than you think!" *Genome biology*, vol. 12, no. 8, p. 125, 2011.

[11] X. Yang, K. S. Dorman, and S. Aluru, "Reptile: Representative tiling for short read error correction," *Bioinformatics*, vol. 26, no. 20, pp. 2526–2533, 2010.

[12] H. Prokop, "Cache-oblivious algorithms," Master's thesis, Massachusetts Institute of Technology, USA, 1999.

[13] F. Rønn, "Cache-oblivious searching and sorting," Master's thesis, University of Copenhagen, Denmark, 2003.

[14] X. Yang, S. P. Chockalingam, and S. Aluru, "A survey of error-correction methods for next-generation sequencing," *Briefings in bioinformatics*, vol. 14, no. 1, pp. 56–66, 2013.

[15] M. Molnar and L. Ilie, "Correcting illumina data," *Briefings in bioinformatics*, bbu029, 2014.

[16] Y. Liu, J. Schröder, and B. Schmidt, "Musket: A multistage k-mer spectrum-based error corrector for illumina sequence data," *Bioinformatics*, vol. 29, no. 3, pp. 308–315, 2013.

[17] L. Ilie and M. Molnar, "Racer: Rapid and accurate correction of errors in reads," *Bioinformatics*, btt407, 2013.

[18] J. T. Simpson and R. Durbin, "Efficient de novo assembly of large genomes using compressed data structures," *Genome research*, vol. 22, no. 3, pp. 549–556, 2012.

[19] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, 1970, pp. 107–141.

[20] P. Medvedev, K. Georgiou, G. Myers, and M. Brudno, "Computability of models for sequence assembly," in *International Workshop on Algorithms in Bioinformatics*, Springer, 2007, pp. 289–301.

[21] K. R. Bradnam, J. N. Fass, A. Alexandrov, P. Baranay, M. Bechner, I. Birol, S. Boisvert, J. A. Chapman, G. Chapuis, R. Chikhi, *et al.*, "Assemblathon 2: Evaluating de novo methods of genome assembly in three vertebrate species," *GigaScience*, vol. 2, no. 1, p. 10, 2013.

[22] T. Pan, P. Flick, C. Jain, Y. Liu, and S. Aluru, "Kmerind: A flexible parallel library for k-mer indexing of biological sequences on distributed memory systems," in *Proceedings of the 7th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, ACM, 2016, pp. 422–433.

[23] P. Flick, C. Jain, T. Pan, and S. Aluru, "A parallel connectivity algorithm for de bruijn graphs in metagenomic applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2015, p. 15.

[24] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Engineering*, Springer, 2016, pp. 117–158.

[25] B. Hendrickson and R. W. Leland, "A multi-level algorithm for partitioning graphs.," *In Proceedings of the SC*, vol. 95, no. 28, 1995.

[26] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, IEEE Computer Society, 2015, pp. 1055–1064.

[27] R. Chikhi and P. Medvedev, "Informed and automated k-mer size selection for genome assembly," *Bioinformatics*, btt310, 2013.

[28] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[29] S. Vinga and J. Almeida, "Alignment-free sequence comparison – a review," *Bioinformatics*, vol. 19, no. 4, pp. 513–523, 2003.

[30] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown, "Scaling metagenome sequence assembly with probabilistic de bruijn graphs," *Proceedings of the National Academy of Sciences*, vol. 109, no. 33, pp. 13 272–13 277, 2012.

[31] D. Fimereli, V. Detours, and T. Konopka, "Triagetools: Tools for partitioning and prioritizing analysis of high-throughput sequencing data," *Nucleic acids research*, vol. 41, no. 7, e86–e86, 2013.

[32] S. Chandak, K. Tatwawadi, I. Ochoa, M. Hernaez, and T. Weissman, "Spring: A next-generation compressor for fastq data," *Bioinformatics*, 2018.

[33] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants," *Nucleic acids research*, vol. 38, no. 6, pp. 1767–1771, 2009.

[34] Ł. Roguski, I. Ochoa, M. Hernaez, and S. Deorowicz, "Fastore–a space-saving solution for raw sequencing data," *Bioinformatics*, vol. 1, p. 9, 2018.

[35] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[36] Ł. Roguski and S. Deorowicz, "Dsrc 2 - industry-oriented compression of fastq files," *Bioinformatics*, vol. 30, no. 15, pp. 2213–2215, 2014.

[37] J. K. Bonfield and M. V. Mahoney, "Compression of fastq and sam format sequencing data," *PloS one*, vol. 8, no. 3, e59190, 2013.

[38] A. Dutta, M. M. Haque, T. Bose, C. V.S. K. Reddy, and S. S. Mande, "Fqc: A novel approach for efficient compression, archival, and dissemination of fastq datasets," *Journal of bioinformatics and computational biology*, vol. 13, no. 03, p. 1 541 003, 2015.

[39] F. Hach, I. Numanagić, C. Alkan, and S. C. Sahinalp, "Scalce: Boosting sequence compression algorithms using locally consistent encoding," *Bioinformatics*, vol. 28, no. 23, pp. 3051–3057, 2012.

[40] Z.-A. Huang, Z. Wen, Q. Deng, Y. Chu, Y. Sun, and Z. Zhu, "Lw-fqzip 2: A parallelized reference-based compression of fastq files," *BMC bioinformatics*, vol. 18, no. 1, p. 179, 2017.

[41] D. C. Jones, W. L. Ruzzo, X. Peng, and M. G. Katze, "Compression of next-generation sequencing reads aided by highly efficient de novo assembly," *Nucleic acids research*, vol. 40, no. 22, e171–e171, 2012.

[42] G. Benoit, C. Lemaitre, D. Lavenier, E. Drezen, T. Dayris, R. Uricaru, and G. Rizk, "Reference-free compression of high throughput sequencing data with a probabilistic de bruijn graph," *BMC bioinformatics*, vol. 16, no. 1, p. 288, 2015.

[43] C. Kingsford and R. Patro, "Reference-based compression of short-read sequences using path encoding," *Bioinformatics*, vol. 31, no. 12, pp. 1920–1928, 2015.

[44] R. Patro and C. Kingsford, "Data-dependent bucketing improves reference-free compression of sequencing reads," *Bioinformatics*, vol. 31, no. 17, pp. 2770–2777, 2015.

[45] I. Numanagić, J. K. Bonfield, F. Hach, J. Voges, J. Ostermann, C. Alberti, M. Mattavelli, and S. C. Sahinalp, "Comparison of high-throughput sequencing data compression tools," *nature methods*, vol. 13, no. 12, p. 1005, 2016.

[46] Y. Liu, Z. Yu, M. E. Dinger, and J. Li, "Index suffix–prefix overlaps by (w, k)-minimizer to generate long contigs for reads compression," *Bioinformatics*, 2018.

[47] S. Canzar and S. L. Salzberg, "Short read mapping: An algorithmic tour," *Proceedings of the IEEE*, vol. 105, no. 3, pp. 436–458, 2017.